Towards a Standard for Subject-oriented Modelling and Implementation

Matthes Elstermann *Editor*

Towards a Standard for Subject-oriented Modelling and Implementation

Working Document

Egon Börger
xyz

Stephan Borgert
xyz

Matthes Elstermann
Karlsruhe Institute of Technology, Germany

Albert Fleischmann
InterAktiv Unternehmensberatung, Pfaffenhofen a.d. Ilm, Germany

Reinhard Gniza
xyz

Herbert Kindermann
xyz

Florian Krenn
xyz

Thomas Schaller
Hochschule Hof, Germany

Werner Schmidt
Technische Hochschule, Ingolstadt, Germany

Robert Singer
FH JOANNEUM–University of Applied Sciences, Graz, Austria

Christian Stary
Johannes Kepler Universität, Linz, Austria

Florian Strecker
xyz

André Wolski
Technische Universität, Darmstadt, Germany

Conny Zebold
Technische Hochschule, Ingolstadt, Germany

Typeset by the authors

Production and publishing: XYZ

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

Short contents

Commands for Review process

For managing changes the package change is used. Following commands are essential:
Mark text without id:
This is the way text is marked

Add todo without id:
Here is some text Here is some text

The original todo note without changed colours.
Here's another line.

Add todo without id and inline:
Here is some text

Here is some text

Add text with id:
[id=AF, comment=remarks to added text]Added text

Delete command with id:
[id=WS, comment=example for delete]Here is the text to be deleted Here is some text

Replace text with id:
Here is some text Here is some text

highlight text with id:
Here is some text
Add todo with fancy line but without id:
Here is some text Here is some text

The original todo note.
Here's another line with a fancy line

For todo commands ids are not allowed. For the add, highlight and Documentation for using the changes package can be found:
http://ctan.ebinger.cc/tex-archive/macros/latex/contrib/changes/changes.english.pdf
The ids and colors for the various reviewers can be found in the preamble (line 29-33) of the text file.

Preface

The preface has to be more detailed

Recent years have seen significant increases in the scope and complexity of digital systems - cf. hhe uprise of Cyber-Physical Systems. Developers, providers, and users have recognized the necessity to devote greater attention to the adoption of innovative technologies. It requires a novel type of preparedness, in particular to the process of design and dynamic adaptation to emergent systems. Since subject-oriented thinking and design supports a human-oriented way to structure and imoplement complex systems, its approach to modeling and execution should be 'standardized', as different teams or modelers could interpret subject oriented means of representation differently. Since subject-orientation is a behavior-centered approach to system understanding and development, non-standardized application could easily lead to non-intended behavior of systems or their components.

However, standardization efforts requires essential steps, among them (see also Figure 1)the process into seven steps.

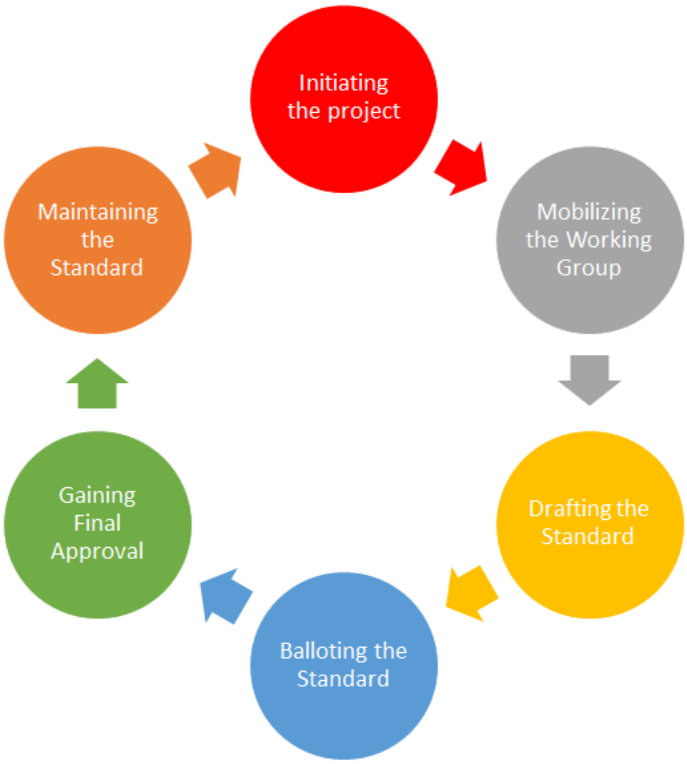


Figure 1: Standardisation Process

1. initiating the project

   Although we have started in 2012 (Fleischmann et al., 2012) the needs for a standard has been identified 2 years ago. We started documenting (Referenz zu paper bei S-BPM 2019?) experiences from providers, users, researchers and system developers. Technological advancements in model execution have led to standardize the semantics, in order to reflect the novel opportunities. Once a need is identified, a proposal to create a new standard has to be put forward.

2. Mobilize a working group

   Standards are created or reviewed by experts in the relevant field. They include researchers, providers, users and communities of practice, who form into a some technical committee, termed the Standardization Gang for subject orientation.

3. Balloting the standard by public review

   The technical committee conducts preliminary research and creates a draft outline of the new standard. Much of this initial work can be done remotely or in sub groups, however, needs to be consolidated at some point in time. This is where we are today when providing this edition of consolidated standard inputs.

4. Gain approval

   Once a draft is written, it requires approval for public review. This consensus is required in order to progress any further. The next step is to include all feedback and create a revised document for final public review. Thereby, anyone is welcome to provide feedback to improve the quality and ensure all relevant areas and perspectives are captured.

5. Publish and Maintain

   After public review, the standard goes back to the technical committee to make amendments it deems necessary based on the feedback received. The committee then approves the final version of the standard. After the revised standard receives that final approval from the technical committee, it is officially released. System developers or providers may incorporate it into their practice.

The current version is intended for researcher who work in the area of software engineering and business process management. It helps understanding the approach through in-depth provision of the practical and theoretical aspects of subject oriented modeling and implementing systems.
The book is a collection of all the essential aspects of subject oriented modeling and programming. Many parts of this documents are reprints of various books and papers. Table 1 shows the sources for the various chapters and sections.
In chapter 1 an overview is given to the subject orientated language PASS and the methods which are used to describe its structural and execution semantics. OWl (Web Ontology Language) [www] is used to describe the structural semantics and Abstract State Machines (ASM) [BR18], [BS03] is applied to specify the execution demantics precisely.
In chapter 2 the structure of PASS specification is considered in detail and its

semantic is defined precisely in a formal way using OWL and ASM. The semantic of the execution of PASS models is described in chapter 3. The execution semantic uses coreASM which is an executable extension of the ASM method. This means that models described in PASS can be executed by a coresponding interpreter.
Chapter 4 shows how the abstract implementation independent PASS models can be implemented using software components, physical components or human. A formal language is described how the right ressources are assigned to the entities of the model.
In the last chapter some additional aspects of the subject oriented modeling approach is considered. Thes aspects extend the kernel which was defined in chapter 2, 3 and 4.
The appendix contains the details of the formal semantic of PASS. Appendix A contains the complete Ontology, Abbendix B defines the mapping of the ontology to the ASM definition of PASS and Appendix C containes the complete formal execution semantics.

contact publisher for clarifying copyrights

Chapter Nr.	**Chapter title**	**Source**
1.1 2.1 3.1	Subject Orientation and PASS Informal Description Informal Description of Subject Behaviour and its Execution	Fleischmann, Albert; Schmidt, Werner; Stary, Christian; Obermeier, Stefan; Börger, Egon; *Subject-Oriented Business Process Management*, Springer Verlag Berlin 2012
3.3	ASM Definition of Subject Execution	
4.1	People and Organisations	Fleischmann, Albert; Oppl, Stefan; Schmidt, Werner; Stary, Christian; *Contextual Process Digitalization: Changing Perspectives - Design Thinking - Value-Led Design* Springer Verlag Berlin 2020
5.1	Subjects and Shared Input Pools	Fleischmann, A.; Stary, C., *Dependable Data Sharing in Dynamic IoT-Systems - Subject-oriented Process Design, Complex Event Processing, and Blockchains;* in Proceedings of S-BPM ONE 2019, 11th International Conference on Subject Oriented Business Process Management, editors: Betz, S.;Elstermann, M.; Lederer, M; ICPC published by Association of Computing Machinery (ACM) Digital Library; 2019

5.2	Subject-Phase Model based process specifications	Fleischmann, A., *Activity-Based Costing for S-BPM,* Proceedings of the 5th International Conference S-BPM ONE 2013, Computer and Information Sciences (CCIS), No. 360, editors: Fischer, H. and Schneeberger, J. Springer 2013,
5.3	Hierarchies in Communication Oriented Business Process Models	Elstermann, M and Fleischmann, A., *Modeling Complex Process Systems with Subject Oriebted Means,* in Proceedings of S-BPM ONE 2019, 11th International Conference on Subject Oriented Business Process Management, editors: Betz, S.;Elstermann, M.; Lederer, M; ICPC published by Association of Computing Machinery (ACM) Digital Library; 2019
5.4	Business Activity Monitoring for S-BPM	Schmidt, W.; Fleischmann, A.; *Business Process Monitoring with S-BPM,* Proceedings of the 5th International Conference S-BPM ONE 2013, Computer and Information Sciences (CCIS), No. 360, editors: Fischer, H. and Schneeberger, J. Springer 2013,
5.5	Subject Oriented Project Management	Albert Fleischmann, Werner Schmidt, Christian Stary; *Subject Oriented Project Management,* published in SEAA '14: Proceedings of the 2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications, IEEE Computer Society, 2014
5.6	Subject Oriented Fog Computing	Stary, C. ; Fleischmann, A. ; Schmidt, W., *Subject-oriented Fog Computing: Enabling Stakeholder Participation in Development,* Proceedings of the 4th IEEE World Forum on Internet of Things (WF-IoT), Singapure IEEE Xplore Digital Library, DOI 10.1109/WF-IoT.2018.8355167, 2018

5.6	Activity Based Costeing	Zehbold, C.; Schmidt, W.; Fleischmann, A., *Activity-Based Costing for S-BPM,* Proceedings of the 5th International Conference S-BPM ONE 2013, Computer and Information Sciences (CCIS), No. 360, editors: Fischer, H. and Schneeberger, J. Springer 2013,

Table 1: Main sources of the various chapters and sections

Contents

CHAPTER 1

Overview

To facilitate the understanding of the later sections, we will introduce the concept of subject-oriented process modeling with the modeling language 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 the Parallel Activity Specification Scheme (PASS) as a language for describing processes in a subject-oriented way. It will not a complete description of all PASS features, but it is a first impression and introduction to the principle of subject-orientation. The detailed concepts are defined in later chapters.

The Term Subject in Subject-Orientation

The term *subject* has manifold meanings, depending on the discipline or domain it is used in. In philosophy, a subject is an observer and an object is a thing observed. In the grammar of many natural languages, the term subject has a slightly different meaning. "According to the traditional view, the [grammatical] subject is the doer of the action (actor) or the element that expresses what the sentence is about (topic)." [Kee76]. In the English language, the term subject can also be used as a synonym for *topic* as, e.g., the *subject* of an e-mail.

However, it is the first part of the second meaning that subject-orientation as a paradigm and PASS as a modeling language refer to in their concept. The term *subject* corresponds to the concept of *doer of an action*[1].

1.1.1 Subject-driven Business Processes

In principle, subjects are the active entities in a specific process context and have a specific behavior. A specification of a subject does not say anything about the technology used to execute the described behavior. This is different to other approaches, such as multi-agent systems, that usually imply a technical/software solution consisting of executable source code.

[1] In contrast to e.g. talks to e.g. ontology description languages, like RDF (see section 1.2), where term *subject* means the topic what the "sentence" is about

Subjects communicate with each other by exchanging information in form of *messages*. Messages have a name and a payload. The name should express the topic or content of a message informally. The payload is the data (business objects) transported.

During the execution of a (business) process, a subject sends messages to other subjects, expects messages from other subjects, and executes internal actions, such as calculating a price, storing an address, etc.. All these activities are done in sequences as no single processor or human can really multi-task. The possible sequence is 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

Subject-oriented system development has been inspired by various process algebras (see e.g. [FSS+12a], [Mil89], [Mil99], [Hoa85]), by the basic structure of nearly all natural languages (Subject, Predicate, Object) and the systemic sociology developed by Niklas Luhmann [Ber11],[Luh84] and Jürgen Habermas [Röm15], [Hab81]. In the active voice of many natural languages a complete sentence consists of the basic components subject, predicate and objects. The subject represents the active element, the predicate the action and the object is the entity on which the action is executed. According to the organizational theory developed by Luhmann and Habermas the smallest organization consists of communication executed between at least two information processing entities (Note, this is a definition by a sociologist, not by a computer scientist) [Luh84]. Figure 1.1 summarizes the different inspirations of subject orientation. The enhancements, such as the graphical notation, constitute the subject oriented approach and will be detailed in the following sections.

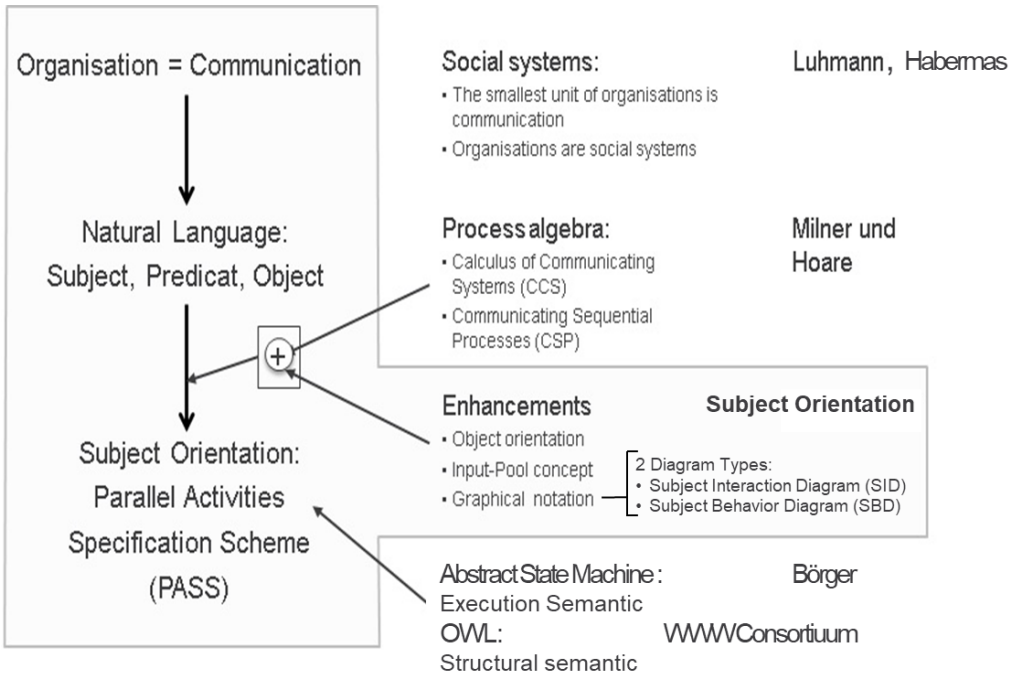


Figure 1.1: Fundamentals of Subject Orientation

Particularly we describe how these various ingredients are combined in an

orthogonal way to a modeling language for scenarios in which active entities have a certain importance. This language is called Parallel Activity Specification Schema (PASS).

1.1.2 Subject-Orientation

Consequently the term Subject-Orientation may be defined as following:

Subject-Orientation is a modeling or description paradigm for processes that is derived from the structure of natural languages. It requires the explicit and continuous consideration of active entities within the bounds of a process as the conceptual center of description. Active entities (subjects) and passive elements (objects) must always be distinguished and activities or task can only be described in the context of a subject. The interaction between subjects is of particular importance and must explicitly be described as exchange of information that cannot be omitted [Els19]

1.1.3 PASS

All the previously given concepts have been adapted into a simple graphical notation, that is the *Parallel Activity Specification Schema* - or PASS . Following the principle of subject-orientation,the main concept of PASS is that, first the interaction between Subjects is described and only afterwards the behavior of subjects can be modeled individually. The details are given in the following sections.

1.1.4 Modeling Subject Interaction

We introduce the basic concepts of subject-oriented process modeling with a simple *order process*. In that 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 1.2 shows the communication structure of that process. There all involved subjects and the exchanged messages can easily be discerned.

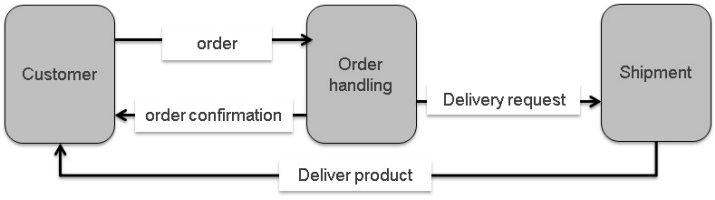


Figure 1.2: The Communication Structure of a simple Order Process

It is assumed that each subject has a so-called *input pool* which is basically its mailbox for receiving messages. This input pool can be structured via rules according to the requirements in given process context. 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 should have their meaning expressed by their name. A formal semantics is given by their use and the data which are transported with a message.

1.1.5 Modeling Subject Behavior

Figure 1.3 depicts the behavior of the subjects "customer" and "order handling".

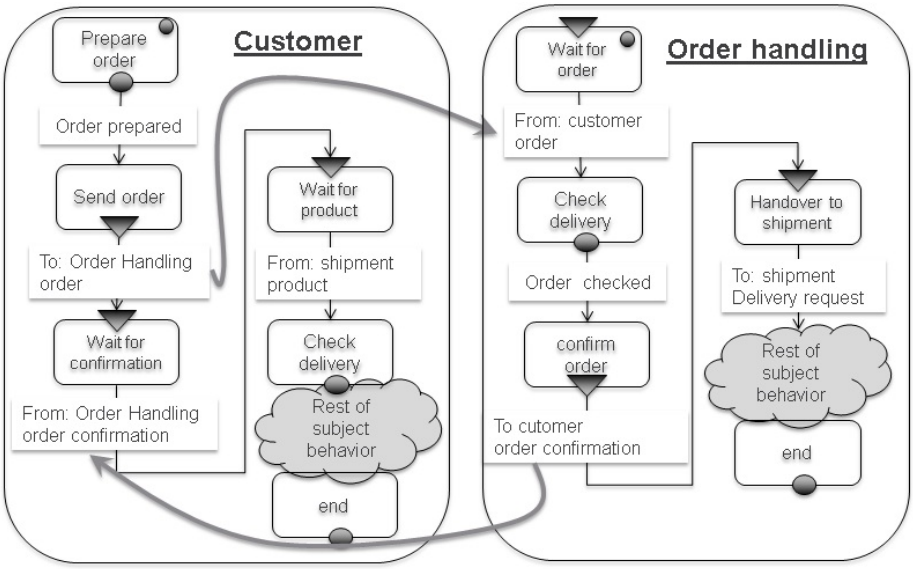


Figure 1.3: The Behaviors of the Subjects from 1.2

In the first state (Do State) of its behavior, the subject "customer" executes the *internal function* "Prepare order". When this function is finished the transition "order prepared" follows. It basically states the condition that must be fulfilled in order to continue. In the succeeding (Send) 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" is allowed to go into the (Receive) 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 and the according receive transition condition is fulfilled. On arrival, the subject removes the message from the input pool and follows the transition into state "Wait for product" and so on.

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

In summary, the behavior of each subject describes in which order it sends and receives (expects) 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. These data aspects of a subject are described in section 1.1.6.

1.1.6 Subjects and Data

Up to now, we did not explicitly mention data or rather the concept of data, even though they are necessary to get complete sentences comprising subject, predicate (verbs), and object. In subject-orientation (data-)objects have their place[2]

[2]Theoretically, the means subject-oriented modeling and object-oriented modeling are 100% compatible. In regards to natural languages it must be understood that pure object-oriented means are akin the passive sentence structures where the subject may be omitted, while subject-oriented descriptions follow the structure of active sentences, that are much easier to understand.

It is assumed that each subject posses an individual "database" to allow to store objects of arbitrary complexity. Similarly, messages may have a data *Payload* representing the transport of data between subjects. Figure 1.4 displays how subjects and objects are connected in the classical way. The internal function "prepare order" uses or manipulates internal data to prepare the data for the order message. This order data object is sent as the *payload* of the message "order".

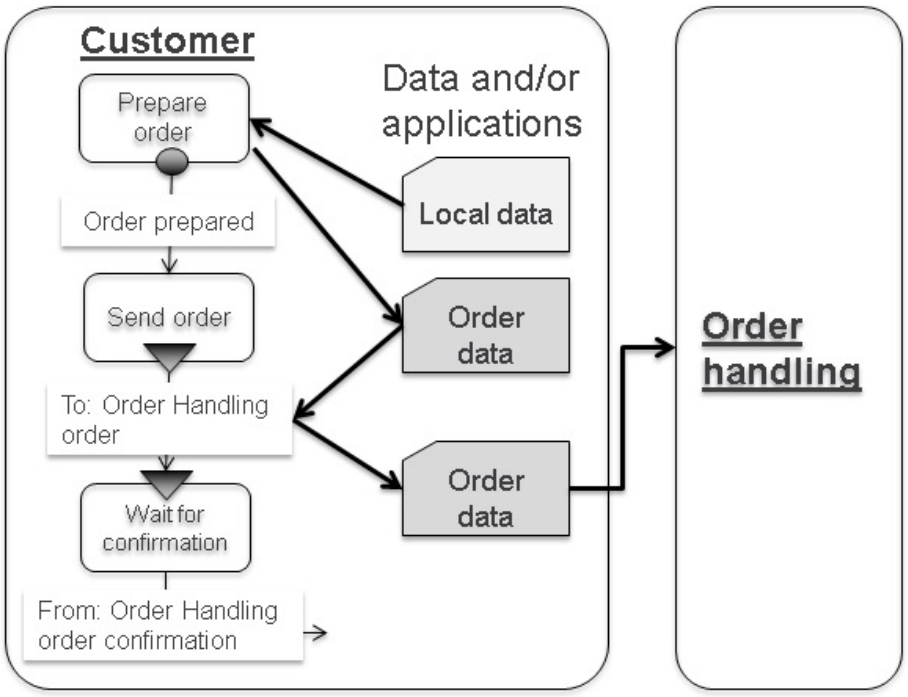


Figure 1.4: Subjects and (Data-)Objects

Modeling wise, internal functions in a subject can be described to be executing the (passive) methods of a specific data-object or as function calls implemented in a service, if a service-oriented architecture is available.

The functions of send and receive states are assumed to always have an additional method that transfer data between messages and internal data store of a subject. If a message is sent, the method writes data values into to the sent message, and if a message is received the corresponding method is used to copy the received data into the subject's data store. In other words, subject either use synchronous services as an implementation of functions, or asynchronous services that are implemented through other subjects or even through complex processes consisting of multiple subjects themselves. Consequently, the concept Service Oriented Architecture (SOA) is complementary to S-BPM: Subjects are the entities which use the services offered by SOAs.

More on data modeling in the context of subject-orientation can be found in Section 2.5.

1.2 Introduction to Ontologies and OWL

This short introduction to ontology, the Resource Description Framework (RDF), and the Web Ontology Language (OWL) will help to get an understanding of

their usage as definition technologies for Subject-Oriented Modeling and Implementation standard in the form of the PASS ontology that is outlined in sections 2 and 3.2.

Ontologies are a formal collections of knowledge or descriptions about the world. They are being used to structure the knowledge of various domains using taxonomies and classification networks. They often use structures similar to human languages, where nouns in statement-sentences represent objects or classes of objects and the verbs represents relations between the objects of classes. (e.g. *"PASS" "is-a" "process modeling language"*)

In computer- and information science, an ontology is a collection or representation of formal names, class definitions, properties, relations between them, and according entities with their relations - usually in regards to a specific thematic domain.

1.2.1 RDF and OWL

The Resource Description Framework (RDF)[www15a] 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 1.5 shows an RDF graph.

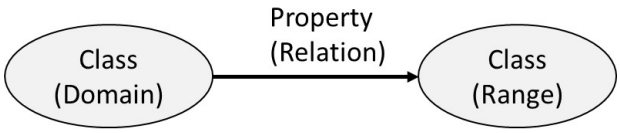


Figure 1.5: RDF graphic

RDF is based on the idea of making statements about resources (in particular web resources) in expressions of the form subject–predicate–object, known as triples. The subject denotes the resource, and the predicate denotes traits or aspects of the resource and expresses a relationship between the subject and the object. In the context of ontology, the term subject expresses what the sentence is about (topic), it must not be confused with term subject in the context of Subject-Orientation (see 1.1).

For describing ontologies several formal languages have been developed. One widely used language is OWL (Web Ontology Language)[www16], which is based on, or a kind of extension to the Resource Description Framework (RDF).

OWL introduces and allows to use the modeling mechanisms of classes, properties, and instances. Classes represent terms also called concepts. Classes have properties and instances are individuals of one or more class.

A class is a type of thing. A type of "resource" in the RDF sense can be a 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 (super class). 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.

OWL 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

Where the OWL/RDF ontology concepts will be used to define the static structure of subject-oriented process models. The execution semantics will be specified using the concept of *Abstract State Machines*. This section will give a short introduction into this topic in order to help the understanding of their usage in section 3.2.

An abstract state machine (ASM) is a state machine[3] 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). In simpler terms, where in standard state machines the states are explicitly defined (e.g. *state01*, *state02*), in an ASM the states or only implicitly or abstractly defined (e.g. *"a state where a data value is x"*).

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 languages, i.e., rules of the (symbolic) form:

$$\boldsymbol{if}\, Condition \boldsymbol{then}\, ACTION$$

[3]State Machines and therefore also abstract state machines are both formal description concepts in theoretical computer science used to describe or define the execution order of a system - e.g. a software program

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 be performed in any given state, when the indicated condition holds true - the *CONDITION* there indirectly defines that state.

The unrestricted generality of the used notion of *CONDITION* and *ACTION* is guaranteed by using so-called Tarski structures as ASM-states, i.e., arbitrary sets of arbitrary elements with arbitrary functions and relations defined on them. These structures are not necessarily fixed, but are themselves updatable by rules of the form above.

An (asynchronous or distributed) ASM consists of a set of agents, each equipped with an individual set of rules in the above given form, called its program. In principle, every agent can evaluated and , if applicable, execute all its rules at any time in one step (in any arbitrary state). In contrast, an ASM, that has only one agent, is called sequential ASM. In general, each agent has its own "time" to execute a step, and a step is independent of the steps of other agents. However there in special cases, multiple agents can also execute their steps in a synchronous manner.

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

$$\textbf{if } Cond \textbf{ then } M1 \textbf{ else } M2$$

instead of the equivalent ASM with two rules:

$$\textbf{if } Cond \textbf{ then} M1 \textbf{ if not } Cond \textbf{ then } M2$$

Another notation used below is

$$\textbf{let } x \ = \ t \textbf{ in } M$$

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

For details of a mathematical definition of the semantics of ASMs which justifies their intuitive (rule-based or pseudo-code) understanding, we refer the reader to the AsmBook Börger, E., Stärk R. Abstract State Machines. A Method for High-Level System Design and Analysis. Springer, 2003 [BS03].

CHAPTER 2

Informal Descriptions of PASS

This chapter explains the principle ideas of the structure and subsequent execution of the graphical, subject-oriented process modeling language that is the Parallel Activity Specification Schema (PASS). It is intended for people unfamiliar with PASS as an introduction to the most important terms and concepts of PASS.

First the principle Structure of PASS and it's two diagram types are introduced in sections 2.2 and 2.3. Afterwards the idea of executing PASS is detailed in section 2.6.

The formal definitions of PASS's structure and its execution are following in the next chapter (Chapter 3).

2.1 Principle Structure

In contrast to classic process description means, PASS process models do not consist of a single diagram graph but may consist of multiple diagrams of principle two types that together form the process model.

Each model consist of a **Subject Interaction Diagram (SID)** that defines the *Subjects* in a process or process system and their principle interaction means in the form of messages.

The second diagram type of PASS are the **Subject Behavior Diagrams (SBD)**, or **Subject Behavior** for short, that should exist at least once for every standard **Fully Specified Subject** in an SID. Next to main or **Subject Base Behavior** diagram, a subject may be associated with other, specialized behavior diagrams (See Section 2.4).

There is a one-to-many (1:n) relationship between SID and *its* SBDs. In a coherent PASS model, the content of the SBDs must follow the specifications of the SID.

Both diagram types use a heterogeneous set of symbols!

The principle idea and elements of both diagrams are described in the following sections.

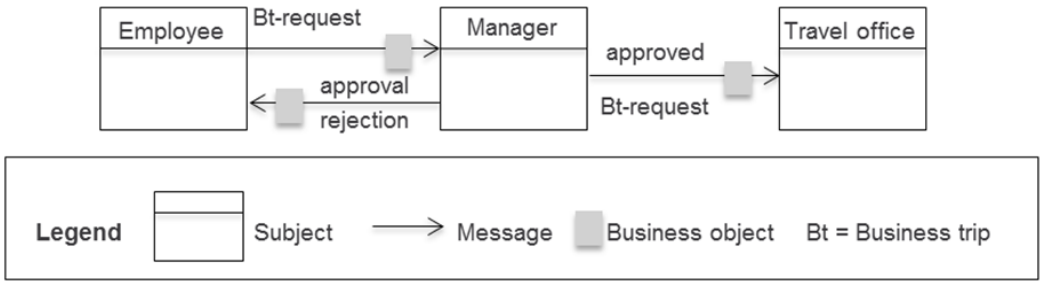


Figure 2.1: Subject Interaction Diagram for the process 'business trip application'

2.2 Subject Interaction

2.2.1 Subject, Messages, and their Interaction

As already detailed previously, **Subjects** defined in a Subject Interaction Diagram, represent an active entity in a process context. This consistency should also be reflected in its name or label and differ it from the passive Objects in the SID or the Activities in the SBD. Note that a specification of a subject does not say anything about the technology used to execute its corresponding behavior.

Subjects communicate with each other by exchanging **Messages**[1]. 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 (its SBD).

In general, there are two principle types of subjects:

- **Fully Specified Subjects**
- **Interface Subjects**

Both come in the variant of:

- **Multi-Subjects**
- **Single-Subjects**

Fully Specified Subjects

The **Fully Specified Subject** is the standard subject type. Fully specified subjects consist of the following components:

- **Subject Behavior Diagram (SBD)** — to specify its behavior in the modeled process context. 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.

[1] The official term for a Message as a model element is called **Message Specifications** - usually both terms are used as synonyms

- **Data Definition** — Each subject is assumed to have a private data storage that contains e.g. business objects relevant in a modeled process context. Examples for contained data 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 other complex data structures themselves - the exact means to specify the structure and types of the internal data store of a subject are identical to the means to specify the payload of massages (**Payload Description**) as described later (See also section 2.5).

- Outgoing **Message Exchanges** (Sent Messages) — Messages which a subject sends to other subjects. Each Message has a name and may transport some data objects as a payload.

- Incoming **Message Exchanges** (Received Messages) — Messages received by a subject. The values of the payload objects are copied to business objects of the receiving subject.

- **Input Pool Restrictions** and **Handling Strategies** — Modeling with PASS assumes that during execution of the process, Messages sent to a subject are deposited in the **Input Pool** of the receiving subject. In principle the size of an input pool is not restricted. However modeling means

- **hasMaximumSubjectInstanceRestriction** — It may be limited how often a Subject may be instantiated or exist in a given process context and in relation to other subjects in the model (See also *Multi-Subject*)

Fully Specified Subjects are also called *Internal Subjects* - as in being defined inside a given model - in contrast to *Interfaces Subjects* that are also known as *External Subjects*.

Interface Subjects

Interface Subjects are used to model interfaces to other process systems. They neither have a behavior, nor a data definitions in a given model, but they can be target or origin of Message Exchanges and may be restricted in their instantiating.

Interfaces Subjects are used when the behavior of a subject is either unknown, if they are not of importance to a model, e.g., if they are too simplistic or modeling would require more effort than the benefit it would bring. Alternatively, the behavior of an interfaces subject is defined in a separate process system models which it may *reference*. In that case their behavior is modeled *externally* in another model - hence the older, alternative term of *External Subject*.

Note: The "other" process models may represent a follow-up or a sub-processes. The exact meaning is up to the modeler, PASS does not explicitly defines on or the other.

Multi-Subjects

Both, Fully Specified as well as Interfaces subjects, may be defined to be **Multi-Subjects** . Multi-Subjects are subjects that may be instantiated multiple times within a process context, implying that there may be several actors executing copies of the same behavior.

This is useful if, e.g. in a process model several identical subjects exist that all do the same principle task in parallel to increase the throughput. Another example would be a procurement process where bids from multiple suppliers are solicited and all suppliers in principle do the same thing - review the request and provide an offer. Once three offers have been received (from the supplier multi-subject), one is selected and actually ordered.

An Interface Subject (A) that is specified as a Multi-Subject may indicate that the whole corresponding process (sub-)system is instantiated multiple times (Multi-Process. Note that a fully specified subject (B) in another model, corresponding to a multi-interface-subject (A) in the first model, that subject B is not necessarily a multi-subject itself if in the context of one instance of the whole (sub-)process there is only one instance of subject B.

Note: While technically feasible to have only Multi-Subjects in a process model, it is assumed to be rarely useful as multiple subjects mostly make sense in the context where there are also single subjects. 1:1 (only single subjects) and 1:n scenarios are most common. n:m scenarios are rather complex and a modeler should be careful to consider if such a situation is really the best solution or if one or more simplified (1:m) models fulfil the same purpose.

Note: Technically, the Multi-Subject is the standard variant of a subject without any restrictions to number of time it may be instantiated in a given process context, in contrast to the *Single Subject*.

Single-Subjects

Single Subjects are all subjects, that are *restricted* to be instantiated only once in the context of a given process model (**hasMaximumSubjectInstanceRestriction**). They are used if for the execution of a subject a resource is required which is only available once.

Because many modelling tools have Single-Subjects as the default modeling setting for subjects, single fully specified subjects can be considered as standard subjects.

2.2.2 Messages and Payloads

As stated, Subjects communicate with each other by exchanging **Messages** . Messages have a name and a (data) payload (**Message Payload**). The name should express the meaning and content of a Message informally. 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[2], but ideally also making clear that a Message is a passive (data-) Object or piece of Information. In the sample 'business trip application', therefore, the Messages are referred to as 'business trip request', 'rejection', and 'approval'.

In a model, Message Specifications define the principle information types or classes of data/information that can be send during execution. The default data payload of a Message is empty — the creation of a Message as well as its sending and reception is already an information that can be evaluated during execution. However, the payload specification can further define the information or data transported with the Message.

[2] A good Message name is akin to a good "subject"-description in an e-mail, giving the recipient a short summary of its content or meaning.

- (formal) **Data Object Definitions** — If the Message and its payload represent digital data to be transmitted within a IT system or similar means, that payload can be described with the means for classical data structure definitions.

- **Physical Object Definitions** — If a PASS process model is not used as an instruction for an IT-System, but as a general description model for real life circumstances, Message are not necessarily data objects but can rather also represent real elements, such as the transport of a physical good in an ordering process. Such an object can be described as the payload of a Message but non-further specified natural language text or photos, etc..

If the described Message Payload is for the former, a digital data object, the Messages serve as a container for the information transmitted. The payload may contain any number of data entries of two principle types:

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

- Complex Data Types/ (Business) Objects[3]). 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.

There are many different means or technologies to specify passive data objects that allow to define, check and verify the structure of data or data objects — the data type. Next to means in most object-oriented programming languages, often in the form classes, there are other languages such as e.g. XML schema[www04] for XML objects or also the RDF Schema [www15b].

PASS, in principle, allows to use any of those to define the data type of payload elements. However, there is also a build in definition means that allow to specify the existence of simple data field of payloads that are either of a simple data type or of a complex data type that itself must consist of data fields (Also see Section 2.5 on discussion of Data modelling).

2.2.3 Start Subjects

Start Subject are Subjects that start their behavior with a Do or Send state (See later section 2.3.1) and therefore are active in a process context from the beginning instead of requiring a Message from another subject to be initialized

It is advised to have only one Start Subject in a process context in order to have a clear chain of initializing interaction. However, it is not required if a process modeler is aware of and the potential consequences.

[3]Business Objects in their general form are physical and logical 'things' that are required to process business transactions. We consider complex data structures composed of elementary data types, or even other data structures, as logical *business objects* in the context of a business processes model

2.2.4 Message Exchange

In the previous subsection, we have stated that Messages are transferred between subjects and have described the means to model the subjects and Messages. What is still missing is a detailed description of how to describe that Messages can or should actually be exchanged and between whom.

In PASS, a **Message Exchange** is a separate model element that groups a triple sending subject, receiving subject, and Message together and thereby specifying for the involved subjects their communication means in the process context. For a complete Message exchange both subjects and the Message must have been previously defined.

Technically, each Message Exchange is a separate triple. However, in visual PASS modeling often multiple Messages are grouped together on the same Message Connector or arrow that connects two subjects. In that case the arrow with multiple Messages can be considered a list or collection of multiple Message Exchanges (a **Message Exchange List**).

2.2.5 Input Pools and the Synchronous and Asynchronous Exchange of Messages

When it comes to the coordination of information exchange there are two principle concepts: synchronous and asynchronous communication.

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 of its behavior, 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 key aspect of the synchronous method is a close (tight) temporal coupling between sender and receiver. While sometimes necessary, for the execution of a problem this potentially raises problems 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.

The counterpart to synchronous communication is the concept of asynchronous messaging, where a sender can send anytime without regard to the state of the receiver. In that case receiver and sender are only loosely coupled.

For that to work, a Message buffer is needed for each receiving subject, where the Message can be placed until the receiver is able to actually receive a Message and thereby take it out of the buffer. In the context of PASS, this Message buffer is called the **Input Pool**. Each Subject is expected to have such an Input Pool; a kind of mailbox or inbox[4] . For a Subject during execution all Messages in its Input Pool are visible. How or when what Message is taken out by the Input Pool owner can be defined via its behavior. E.g., if multiple Messages of same type are in the Input Pool, only the newest or only the oldest is

By default the Input Pool is unrestricted and expected to allow the transmission of an unlimited number of Messages simultaneously. In consequence, this

[4]Note that an Input Pool is expected to exist **per Subject** — per process specific role — and not per entity that executes a subject. So its not a personal input pool for, e.g, an user. See also Section 2.6.1.

assumption makes asynchronous communication the default mode in any PASS model — however, not the only one possible.

Input Pool Constraints and Handling Strategies

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, and also to allow more detailed modelling of, e.g. synchronous communication, constraints can be placed on the input pool.

The concept of the Input Pool is actually a concept of PASS execution. There is no explicit input pool in a process model itself. Its existence is simply assumed for the execution. However, what can be included in a PASS model, are rules that should be applied to a given subject's input pool — the Input Pool Constraints and the handling strategy that should be applied if a given rule does not allow to place further Messages into an Input Pool.

In principle, an Input Pool has the following restriction types (**Input Pool Constraint**) - see also 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 general 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 (**Sender Type Constraint**). Again, a value of zero means that Messages from a specific subject can only be accepted synchronously.
- Maximum number of Messages of a specific type — For an input pool, it can be determined how many Messages of a specific Message Type (e.g., invoice) may be stored simultaneously in the input pool, regardless of what subject they originate from (**Message Type Constraint**). A specified size of zero allows only for synchronous Message reception for that Message Type.
- Maximum number of Messages with specific identifiers of certain subjects — The combination of the two previous restrictions variants (**Message Sender Type Constraint**): it can be determined how many Messages of a specific type from a particular subject may be stored simultaneously in the input pool. The meaning of the zero value is analogous to the other cases.

In eqch case the Input Pool Restrictionhas a natural numberLimit (**hasLimit**). By restricting the size of the input pool, the ability of sending-subjects to place Messages in the input pool may be blocked at a certain point in time during process execution. If any value is set to zero this only allows for synchronous communication by default. However, if the the limit is higher than zero, a run-time-environment must decide how to handle the input pool. There are four

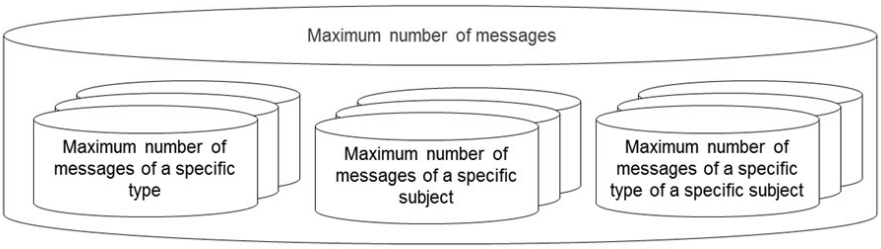


Figure 2.2: Configuration of Input Pool Parameters

strategies (**Input Pool Constraint Handling Strategy**) to handle input in regards to a specific restriction that can be included into PASS.

- **Blocking** the sender — 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.
  (Official name: **InputPoolConstraintStrategy-Blocking**)

- **Delete 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.(Official name: **InputPoolConstraintStrategy-DeleteOldest**)

- **Delete 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.(Official name: **InputPoolConstraintStrategy-DeleteLatest**)

- **Dropping the new Message** after reception — This will allow a sending subject to finish its sending task, but the send message will never be processed by the receiver because it will be deleted upon arrival.(Official name: **InputPoolConstraintStrategy-Drop**)

2.3 Subject Behaviors

As stated: each Fully Specified Subject in a PASS model's Subject Interaction Diagram (SID) has at least one individual Subject Behavior Diagram (SBD) associated with it; it's base behavior[5]. An SBD describes a subject's principle order of activities in a process context; it describes the subject's behavior. Therefore

[5]Macro and Guard Behaviors are described later.

they describe and execution pattern or what to be done. (in contrast to the SID that describes an interaction structure of active entities)

Note: SBD/states describe activities in principle and models necessarily needs to be aware of the execution he is implying. But the model and its execution are two related but different concepts. Be aware of what is talked about!

2.3.1 States, Transitions, and Transition Conditions

In principle, an SBD consist of **States** and **Transitions** used to describe what actions a subject performs in what order. To "be in a state" expresses that the subject is continuously performing the activity described on that state. The activity of state should be denoted at least by its label, but can also be detailed out with a **Function Specification** . One or more outgoing transition arrow denotes the next possible state to be executed as well as the condition (**Transition Condition**) that must be fulfilled in order to follow the denoted path.

Do, Send, and Receive

There are three principle state types in PASS: states that denote the execution of an internal action or function (**Do States**), and communication states to describe interaction with other subjects: This can either be the activity of sending a message to another subject (**Send States**) or the "activity" of waiting for and subsequently taking out a message from the inbox (**Receive State**).

Each state type corresponds with an according transition type: Send States with **Send Transitions** , Receive States with **Receive Transitions** , and Do States with **Do Transitions** .

The Transition Condition to leave a Do-State via a Do Transition usually is that the activity is finished. In case there are multiple outgoing transitions, what the result or outcome of the activity was.

The Transition Condition to leave a Receive State is that a certain Message from a certain sending Subject is in the Inbox of a subject an can be taken out. The actual act of receiving is happening in the moment when following the transition. Before that, the activity of a receive state is, as stated, waiting.

Finally, denoted on a Send Transition, the Transition Condition to leave a Send State is that a specified Message to a specified Receiving Subject has been sent. Before that, the activity of a Send State is concerned with the preparation of the Message.

Figure 2.3 shows the different types of states with the corresponding symbols.

Send and Receive Types

The Transition Condition of Receive and Send Transitions can further be detailed out with a **Receive Type** or respective **Send Type** . These are important to define the interaction of the sending or receiving subject with multi-subjects.

By default the type of both is the standard type (**Send Type Standard** and **Receive Type Standard**). This is intended for interaction with default **Single Subjects** , where in the context of a process model there will be a maximum of one interaction partner subject. In that case, it is expected that a message is send to or received from that interaction partner. If no instance of the partner Subject

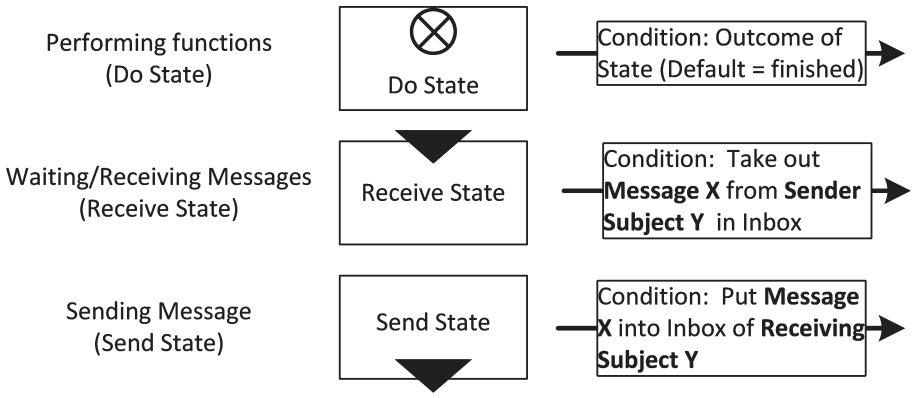


Figure 2.3: Symbols for the principle State types corresponding Transition types in an SBD

exist, such an instance will be created. This is different if the partner Subject is a **Multi-Subject** , implying that there can be multiple communication partners.[6]

In that case, for a Send Transition it can be specified that copies of message-to-be-send either to all known instances or to a **certain number** of instances (**Send Type Multi Send To All** [known] or **SendTypeMultiSendToKnown**). Alternatively, it can be specified that the sending of this message is supposed to go to (or to create) a specified number of new subject instances (**SendTypeMultiSendToNew**)

The other way around, when receiving from a Multi-Instance-Subject, it can be specified for the Transition Condition for a Receive Transition is not the reception of a single Message, but rather multiple copies of the same Message must be received. Either a **specified number** of messages from the known subject instances (**ReceiveTypeMultiReceiveFromKnwon**) or simply each known subject must have answered (**ReceiveTypeMultiReceiveFromAllKnown**).

Start States

In each Behavior Diagram there is one and only one state that is denoted as the **Initial State Of [a] Behavior** or *"Start State"*. And a subject is expected to be in that state upon initialization.

If the Start State of an SBD is a Do State or a Send State the subject is expected to be active upon initialization of the overall process system out of its own accord. This can be denoted in the SID by marking the Subject as Start Subject (See section 2.2.3).

All other Subjects are expected to have a Receive State as the Start State of their SBD and therefore being initialized when another Subject send a message to them for the first time.

End States

Any Do State and any Receive State of an SBD can be denoted to be an **End State** and all state-transition paths in an SBD should lead from the Start State to an End State. There can be multiple End States and a State is allowed to be starting and End State at the same time if the behavior is cyclic.

[6]Reminder: the Multi Subject is a subject without Instantiation Restrictions, while it is the Single Subject is limited to be instantiated only once in a process context. See also Section 2.6.1 for the Concept of Subject Instances

If a Subject Instance is in an End State its execution can be *terminated* permanently by a surrounding process execution system, however termination should only be done if all Subjects in a process context are in an End state during execution. Therefor it is possible to "reactivate" as subject. If the End State is a Receive State this is done via the sending of message received via a Receive Transition leaving that End State. If the End State is a Do State, denoting the the internal activity currently is "Do Nothing", the subject can be reactivated by that subject's carrier (See section 2.6.1) on its own via an outgoing Do Transition or User-Cancel. Alternatively, a Time Transition can be used to trigger a temporal condition to leave either Receive End States or Do End States.

Send States must not be declared to be End States! Being "in a state" means the activity, in that case sending, is not yet finished. However, sending is a explicit activity of informing someone else and should be finished upon the end of process. There simply is no point to model an sending activity without completing it.

Example for Complete Behavior Diagram

Figure 2.4 shows the Subject Behavior Diagrams of the Subjects 'employee', 'manager', and 'travel office' in the sample "Business Trip Application" process who's SID was introduced in figure 2.1:

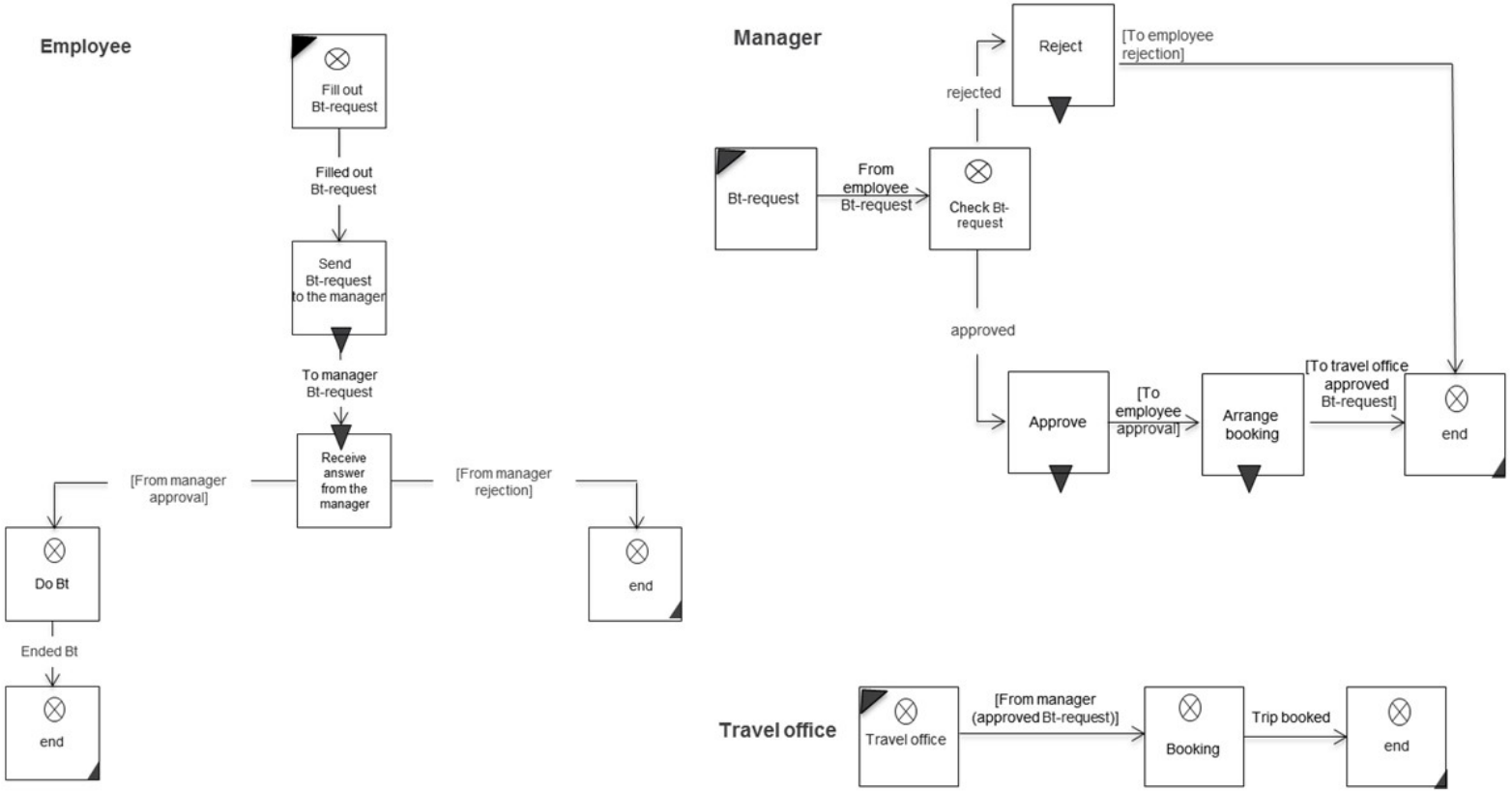


Figure 2.4: Subject Behavior Diagrams for the subjects 'employee', 'manager', and 'travel office' from figure 2.1

In the depicted sample process, employees send completed requests using the message 'business trip request' to the manager's input pool. Upon reception, the manager checks the request and can either reject it and send the according message, or he can approve it. In the later case he must send the 'Approval' reply to the employee and forward the 'Approved Business Trip Request' to the travel office. It does not matter which message is attempted to be sent first. If the send mechanism is successful, the corresponding state transition is executed.

2.3.2 Branching & Priorities

Send States may have only one outgoing Send Transition! In contrast, **Do States** may have multiple outgoing **Do Transitions** and **Receive States** may have multiple outgoing **Receive Transitions**, creating branches or splits in the possible process paths.

Since a subject is not a quantum particle it can be only in one **State** at once, therefore all branching in an SBD is always of exclusive alternative (X-OR) nature; there are no explicit AND- or OR-Splits and Joins in PASS. However, there is a pseudo-AND spit in the form of the **Choice Segment** (See section 2.3.3).

For parallel executed activities (AND-Splits) the subject-oriented **Parallel** Activity Specification Schema (PASS) requires two or more different active entities (Subjects) to be modeled, that can work in parallel. Therefore a Send State/-Transition combo leading to anything but a receive state waiting for direct answer to the sent message is basically an AND-Split while waiting for a corresponding message in a receive state is an AND-Join.

For the case that the conditions of more than one outgoing transition of a state are fulfilled at the same time, each transition has a Priority Number to define a precedence order for choosing one of the valid paths. The PASS standard defines a lower Priority Number to be the higher priority (Priority Number "one" is higher than priority number "five"). If no such clear precedence order is defined and more than one transition condition is true, the path may be chosen a arbitrarily or at random.

2.3.3 Further SBD Elements

In addition to the core states and transitions, a Standard PASS SBD may contain a few other elements: User Cancel Transitions, Time Transitions, Sending Failed Transitions, and Choice Segments.

User Cancel

A **User Cancel Transition** usually denotes the possibility to exit a receive state without the reception of a specific message. Basically, it allows for an arbitrary decision by a subject carrier/processor (See section 2.6.1) to abort a waiting process.

In the example shown in Figure 2.5, the receiving state can also be left to send an additional inquiry message. That message, necessarily, must have been defined in the according SID first.

User Cancel Transitions could also originate from Do States, but here they are not very different from a normal Do-Transition and only emphasise the option to arbitrarily abort the execution of **Do Function Specification**, including the specification of what happens afterwards.

It is also possible to attach a User Cancel to a Send State giving the subject carrier the option not to send the defined message but skip to another state.

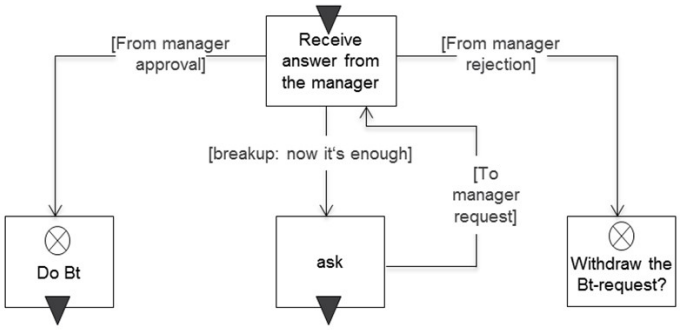


Figure 2.5: Message reception with User Cancel Transition

Time Transitions

Instead of having the decision to skip waiting or abort an activity left to the arbitrary decision of a subject carrier, Time Transitions can be used to model that a state transition should happen based on a time based constraint. There are to principle types of Time Transitions: Timer Transitions and Reminder Transition.

Timer Transitions basically 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 and thus starting the timer.

There are three variations for the Timer Transition based one the unit of time necessary to model them for. First is the **Day Time-Timer Transition** allowing to specify a time-out after a certain amount of Seconds, Minuets, or hours. This is the most basic Timer and shown in figure 2.6 where after waiting three days for the manager's answer, the employee sends another inquiring request.

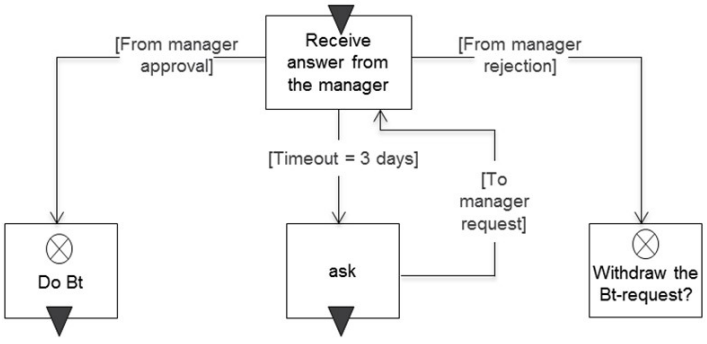


Figure 2.6: Time monitoring for message reception

Due to difference and complication with measuring time calendarial (e.g. leap year concerns, leap seconds, different calendar systems), Time Transition based on years or months need to be different, following the XML standard for noting done time intervals (**Year Month Timer Transition**)

Finally as PASS is often used to model business processes, it is useful to be able to define a time-out after a certain amount of business days have passed, with the definition of business days depending the time the business process is executed and/or the legally binding geographical location of the executing instance (Business Day Timer Transition).

Reminder Transitions are transitions that can be traverses if a certain time based event is triggered, e.g. a certain calendar date has been reached, or frequency condition is met since the last traversal of that transition (e.g. once per day).

As with the Timer Transition, the exact time definition can again either be based on calendarial consideration (**Calendar Based Reminder** Transition) or be based on hours, minuet, or seconds (**Time Based Reminder Transition**).

Reminder Time Transitions are used only in States that will be (re-)visited multiple times. Furthermore, since **Send States** should be executed directly, Reminder Transitions should not originated from them. Reminder Tranistions should only originate from **Receive States** and possibly from **Do States**

Sending Failed Transition

A **Sending Failed Transition** is a technical model element and can only originate from a send state. It is similar to a Time-Out Transition or User-cancel. It is used to denoted an alternative route from a send state if the process execution environment is not able to actually perform the implicated sending activity and thus fulfill the Transition Condition

Choice Segments and Choice Segments Paths

As stated before, the behavior of subjects is regarded as a distinct sequence of internal functions, as well as send and receive activities. Usually the order of the activities is either important and therefor modeled with consideration or the order does not matter and the modeler simply can arbitrarily choose a sequence. However, there are some cases where it is important to allow the order of execution of some states to be up to the choice of the subject carrier.

The freedom of choice with respect to behavior is described as a collection alternative paths in a so called **Choice Segment** . Each **Choice Segment Path** can be modeled to be optional or mandatory to start (**isOptionalToStartChoiceSegmentPath**), as well as separately optional or mandatory to end (**isOptionalToEndChoiceSegmentPath** . This leads to the following combinations.

- Beginning is mandatory/end is mandatory: All states in a path need to be processed to the end.
- Beginning is mandatory/end is optional: The path alternative must be started (first state), but does not need to be finished.
- Beginning is optional/end is mandatory: if the path is started, all states in the path must be completed.
- Beginning is optional/end is optional: path is completely optional

Note that state transitions in a Choice Segment Path must not point to states outside of the path. An alternate sequence starts in its start point and ends entirely within its endpoint.

The execution of a Choice Segment is considered complete when all mandatory to start paths have started and all mandatory to end paths have been entirely processed and have reached the end operator of that path.

Figure 2.7 shows an example for modeling Choice Segments. After receiving an order from the customer, three alternative behavioral sequences can be

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

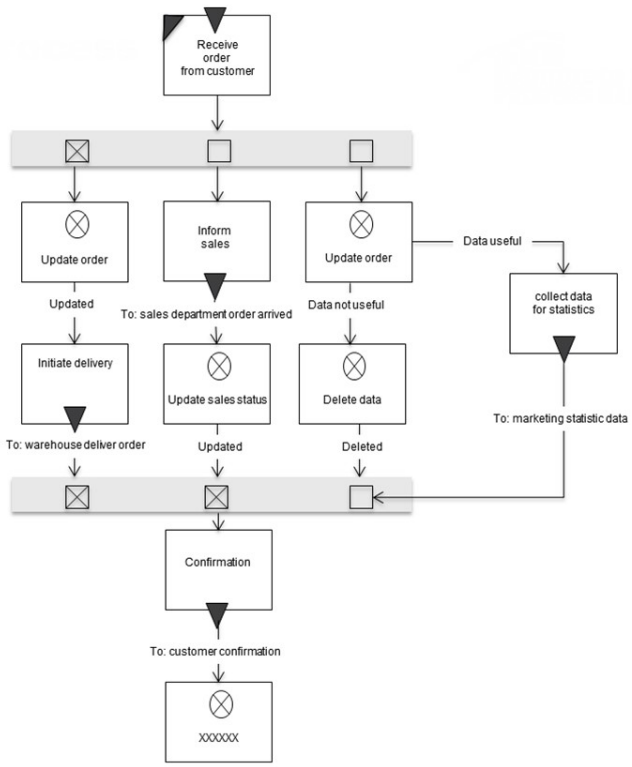


Figure 2.7: Example of a Choice Segment with three Choice Segment Paths

The other two Choice Segment Paths can, but do not have to be, started. However, in case the middle path 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 is completely optional.

For the detailed execution semantic of the Choice Segment refer to section 2.6.2.

The Action-Concept as an SBD element

An **"Action"** , as an element of an SBD-model, simply is a grouping concept that groups a State with all its outgoing valid Transitions. This element rarely has visual representation in visual PASS modeling.

2.4 Multiple Behaviors

The default concept of PASS is, that for a Fully Specified Subject, there is one Subject Behavior Diagram — the **Subject's Base Behavior** . However, for ad-

vanced modeling purposes standard PASS allows to add additional, special-purpose behaviors to a subject. — **Guard Behaviors** or **Macro Behaviors** .

In principle, both, a Guard Behavior, or a (subject specific) Macro Behavior, are individual SBDs that are linked to a subject within the same overall PASS model. The communication capabilities in the additional SBDs are still determined by the same SID.

A subject with multiple behaviors, still always *is* in one State, however that state may be in one of the different behaviors. This implies that the currently *"active"* behavior is changing over the course of the execution of a process model. As a consequence, it is necessary to determine a precedence order between them, to define which Behavior is responsible for a subject's actions if more than one Behavior could potentially be executed at the same time. For that purpose each Behavior **has [a] Priority Number** assigned to it, with the Base Behavior having the lowest priority in form of the highest Priority Number. The concept is, that different behaviors are laying "on top" of each other, forming a kind of "behavior stack" in the order of the priority numbers.

2.4.1 Switching Between Behaviors

At least for an executable Base Behavior, as stated, it is expected that all the states are in a valid path between a Start State and a valid End State.

In Guard Behaviors there does not necessarily need to be an end in an End State, but there can be. Macro Behaviors **must not have an End State** !

Rather, they end in states that refer back to another Behavior to continue the execution there. The referred-to behavior, most likely, is the Base Behavior.

There are two principle ways to refer back to another behavior First there is the specific **State Reference** ; a kind of "state" that refers to a state in another behavior. A transitions leading to such a State Reference "state" is akin to that transition leading to the original state.

The second option is the **Generic Return-To-Origin-Reference** .

Both reference types must not be used in a Base Behavior, they are used under the assumption that the behavior they are used belongs to a model with multiple behaviors and in was entered from a Base Behavior. Either by Macro-Call from a state there, or because the reception of an Interrupt Message has triggered the Guard Behavior.

2.4.2 Guard Behaviors: Exception or Interrupt Handling

Normally, a subject can receive message only in according receive states. However, sometimes there are urgent messages that need to be handled out of order.

For example in the business trip booking process, if there is an additional message from another subject that can make the Employee cancel all the previously made plans, no matter at what point in the process he is.

From communication point of view, this is exemplaryily shown in Figure 2.8 where the concept is, that the newly introduced Service Cancellation Message can arrive arbitrarily at any point in time.

With only one behavior and the classical means, handling of this message can only be done in the normal receive states as an alternatives, as shown in Figure 2.9. Here, the Cancellation message may be handled the earliest after the original request was filled out or after it was send of to the manager, but not after the manager has send the acceptance notice, even if the trip could be canceled afterwards.

Instead, the mechanism of the **Guard Behavior** can be used to model that some specific or all states of a Base Behavior are *"guarded"* by another behavior. A Guard Behavior is a separate SBD that always has a Receive State as its initial state. All Messages that are received in such an initial state of Guard Behavior basically become interrupts or exceptions to the normal flow of the process. Upon the reception of such an Interrupt Message the subject *"jumps"* to that initial state of the Guard Behavior and the process flow continues from there. The flow in a Guard Behavior may either lead to an End State, to a specified state that is referenced in the Guard Behavior, or the process flow can be set to continue where it left off, before switching to the execution of the Guard Behavior (See State Reference or Generic Return-To-Origin-Reference in section 2.4.1). For the whole concept to be specified properly, the Guard Behavior must be set to either guard a specific State (**guardsState**) or a complete Behavior (**guardsBehavior**). Also, the Priority of the Guard Behavior must be higher than that of the guarded Behavior.

Figure 2.10 shows such the multiple Behaviors for the subject Employee.

2.4.3 Subject Specific 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 sub- behavior needs to be described many times.

Instead, a repeated behavior fragment can be defined as a so-called **Macro Behavior** . Such a Macro Behavior then can be called from different states in an SBD as often as required. Thus, variations in behavior can be consolidated, and the overall behavior can be significantly simplified.

To illustrate with an example an simple order processing is being used. Figure 2.11 shows the SBD of a macro behavior that describes the activities to handle customer orders. After placing the 'order', the customer receives an order confirmation; once the 'delivery' occurs, the delivery status is updated. Instead of ending in a normal end state, the Macro Behavior ends in a Generic-Return-To-Origin-Reference, that refers the execution back to the state from which this Macro Behavior was called. This generic approach is necessary, since such a call can happen from multiple times from multiple different states during the execution of the overall process model.

A call to a Macro Behavior is conceptualized similar to an advanced function call or refinement - an addition to one of the states of another SBD. No restrictions have been made regarding the type of the state. A Macro calling state is therefor also called a **Macro State** .

Usually the Macro calling is a replacement for the internal function of a Do-State, or the additional function of Receive and Send states. Calling to the Macro happens before executing the rest. If execution of the Macro Behavior produces results that influence e.g. the choice of a certain branch to continue, this must be reflected in a change in the internal data store of a subject and subsequently be evaluated in the outgoing transitions of the Macro State.

2.5 PASS and Data modeling

PASS is predominantly a modeling language intended to describe processes. However, especially in the context of business processes, interaction with data-

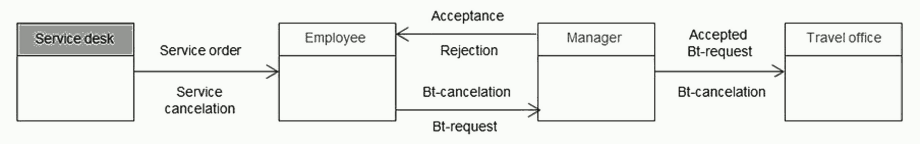


Figure 2.8: Extended SID of the business trip application with additional cancellation message from a Service Desk

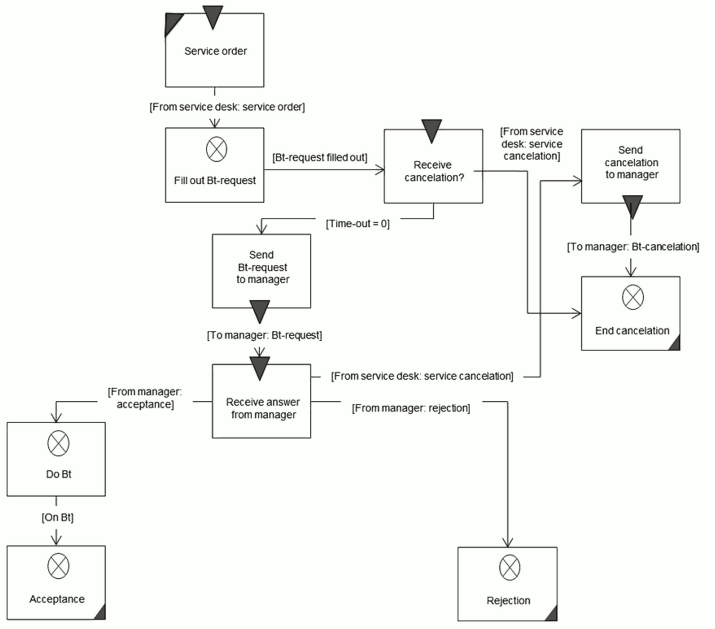


Figure 2.9: Handling the Cancellation Message using only standard SBD constructs

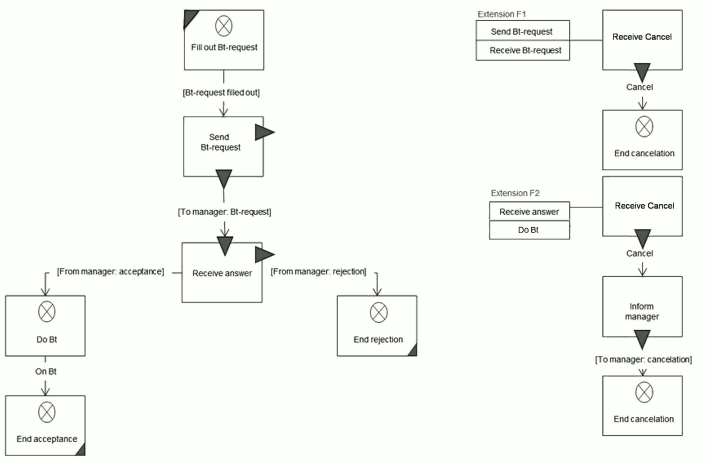


Figure 2.10: Base Behavior and two (separate **Guard Behaviors** of subject 'employee' setup to guard specific States

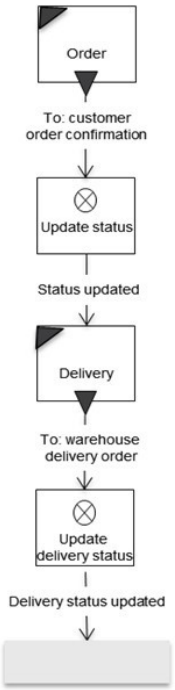


Figure 2.11: Behavior macro class 'request for approval'

objects, often referred to as business objects, are necessary. PASS explicitly calls for modeling these aspects as was already mentioned in section 2.2 or more specific subsections 2.2.1 and 2.2.2.

Subjects as well as **Messages** are assumed to have an individual "data storage" capacity. A Message Specification **contains [a] Payload Description** in form of a **Payload Data Object Definition**. A Subject simply **has [a] Data Definition** . According the principle of subject-orientation, these data stores are **passive** and never do anything on their own, they simply "are there" or "exist".

In principle, any means or technology to define data structures (**Data Object Definition**) and constraints upon them can be used for that purpose, as long as the intended recipient of the model is able to understand it. The technical precondition here is the existence of an exchange format for the data structure definitions that can be used to be stored with the PASS model. Examples are XML-Schema, RDF-Schema (RDFS), or OWL. Due to the model exchange standard for PASS itself being founded on OWL, using that language's means to define data types/classes and constraints upon them and directly integrating them into the model is a suitable approach.

The minimum capability to model the structure of Subjects Data Object Definitions and Message Payloads should be the ability to define the existence of data fields, as well as their data type, their order, and how often they occur in the data structure (e.g. single occurrence or multiple occurrence aka a list). Standard data types for the fields are what can typically be found in any data related descriptions means, e.g. data base query definitions via SQL that contain concepts like integer and floating point numbers, boolean, string, and date/time. Beyond that, the ability to define further, complex data types/classes (**Data Type Definition**), comprised of other data objects, is expected to exists. Figure 2.12 shows such a simple and generic definition approach.

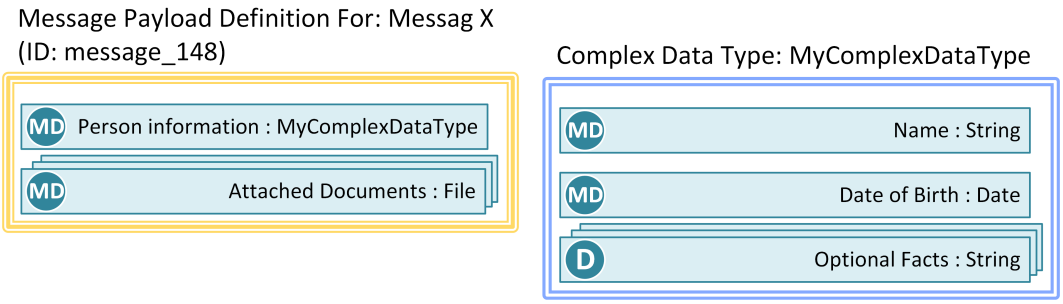


Figure 2.12: Example for a generic Message Payload Definition with field names and data types, including the specification of a Custom Datatype

2.5.1 Data Mapping

In any state of a Subject's Behavior, the data store could be accessed, meaning values or objects can be read or written from and to the data store. The actual access method would depend on the implementation of the an surrounding workflow execution environment/system, beyond the scope of a process model. One setting could be that a subject carrier has principle access to any data field or object of its subject's data store or the Payloads of all incoming and outgoing messages.

However, for automation and/or restriction purposes and under the assumption that access is not given in principle, the PASS standard envision the definition of so called **Data Mapping Function** s that any State may have (**hasDataMappingFunction**).What is "mapped" or matched by a Data Mapping Function is a field from a Subject's Data Store with a field of State Function/s. When return values/object of a function or data field from a message are to be written into the local (writing access) this is called a **Data Mapping Incoming To Local** function. Read Access is given by **Data Mapping Local To Outgoing** functions that map values from a subjects local store with the input parameters of function call or copy them into the Payload of a massage.

Do State s may have both, reading and writing access, since it may be necessary for the performance of the activity specified for the Do state and to store the results. In contrast, **Send States** only are allowed to have read access that maps the data fields to the Payload of the outgoing message (**Data Mapping Local To Outgoing**). Equally, **Receive States** are about receiving data and potentially storing in a subjects data store, hence only **Data Mapping Incoming To Local** Data Mapping Functions are allowed.

In general all read and write activities are expected to be "call-by-value" meaning that only copies of values or objects are transmitted. There is no shared-data-storage explicitly envisioned by Standard PASS. If such is wished for it can be defined as specialized, potentially tool-specific **Function Specification** .

2.6 Informal Description of PASS Execution

PASS as a modeling language is intended to model the dynamic activities of a process. Therefore each process model can be seen as an instruction to execute the process. This implies a certain semantic or meaning that for most parts should be obvious and has already been partially implied or explained in the previous sections describing PASS. However, some points may need to be discussed in order to understand the peculiarities of PASS.

This section describe the execution concept of PASS informally without formal definition. For the formal definition of PASS execution please refer to Chapter 3 Section 3.2.

2.6.1 Principle Ideas of PASS Execution

Models, Subjects, and Instances

In general, a process model can be used as the input to an information system as instruction of how to execute which activities in what order. This abstract definition holds true for any type of execution system, be it a social system in form of an organisation of people that use the model as instructions, or be it a technical (IT) system, usually a so-called workflow engine, that uses the model as program instructions. For the last variant the model must exist in a formal and digital manner, while humans could also cope with a less formal or analog version. In case the process model describes and is intended to be executed by mix of both, a socio-technical system, incorporating digital and analogue elements, the interplay between both elements needs to be clear, and especially the model parts describing the instructions for the digital/workflow elements must be formal, precise, and also consider the interactions of both sides.

Obviously, the same process model can be used as instruction in several **runs** of process execution. In digital workflow systems such a run, or all data it comprises are usually called an **Instance** of the process model, or **Process (Model) Instance**.

The peculiarity of the subject-oriented structure of PASS allows to differentiate further. The instance of PASS process forms the **context** of execution in which individual **Subject Instances** exist. This concept is visualized in Figure 2.13. Subject Instances can be created directly for Subjects with Behaviors that start in a Do or a Send State. Subjects with a Behavior that start in receive states usually are expected to be instantiated upon the sending of an message that is received in an initial Receive state. If not restricted, in principle, each Subject could be instantiated multiple time within one execution context. For that to work properly, every sending action of a subject would need further specifications as detailed in sections on Multi-Subjects (2.2.1) and Send and Receive Types (2.3.1). To keep initial modeling and also execution easier, most PASS modeling tools make the Single Subject and subsequently the Standard Send Type the default[7].

The sub-division between Process Instances and Subject Instances allows the distinction between the termination of the overall process instances or context

[7] There also concept or approaches, most notably Act'n'Connects, that by default call for PASS process models containing only a lone Fully Specified Single Subject that communicates only with Multi-Interface Subjects. Within their approach of the *"Internet of Actors"* there is less focus on the context provided by an overall process model and more on the dynamic creation of the process context managed by the execution IT-Systems with subjects being only loosely coupled.

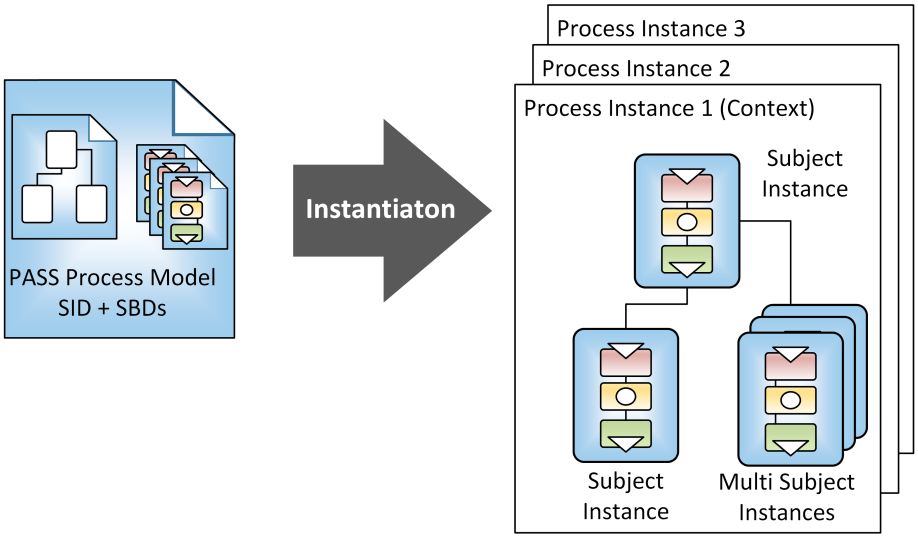


Figure 2.13: PASS Process Models in Relation to Process (Model) Instances and Subject Instances

from the ending of an individual subject instance. As described in the discussion of End States in section 2.3.1, in principle, a subject-is allowed to exit an end state again if modeled so. This only make sense if the overall process context has not yet ended and also requires a concept of when this is not possible anymore. In consequence, the meaning of a Subject Instance being *"in an End State"* of its Behavior is different from a Subject Instance being *"terminated"*. The understanding is, that a Subject Instance can only be terminated when all other Subject Instances in the context of the same process instance are also in an End State.

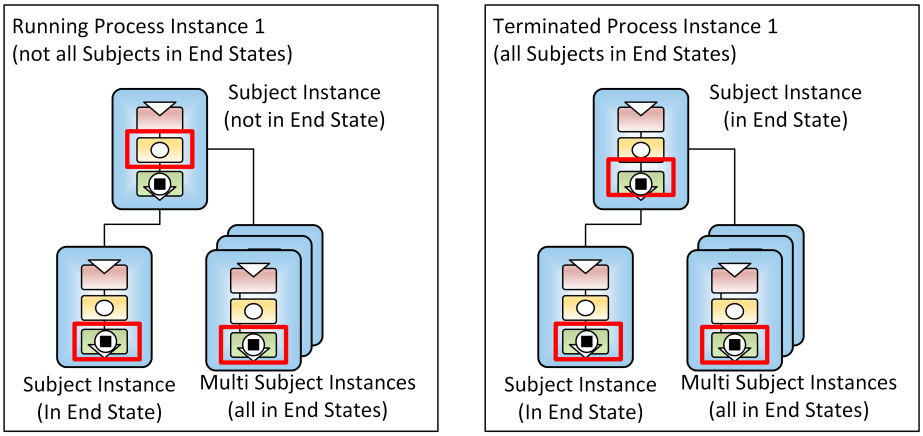


Figure 2.14: Conceptual Depiction of running and terminated process/subject intances

Subjects Instances, Subject Carriers, and Inboxes

As stated in Section 2.2.5 on Input Pool Restrictions, during execution, each Subject Instance is considered to haven Input Pool or Inbox for the Messages. This Inbox is bound to the Subject Instance and not to the person or technical entity that is responsible for executing the instance's behavior.

This entity that *"drives"* the execution of a Subject's Behavior is called the **Subject Carrier** or Processor. The idea of the subject carrier is a general concept and could imply a real person as well as an automated technical system. Within the bounds of an IT-Workflow system, a human being or other real-world entity would necessarily require an according (graphical user) interface (GUI) to make updates to the execution status that is tracked inside the workflow software.

This differentiation between "Subject Carrier" and "Subject" is important to understand PASS and also the terminology of why the model element is called a "Subject". First and foremost, the idea helps to understand that in the Subjects in a process model describe abstract, process specific roles and not the activities of concrete entity. That real life entity, e.g. the manager in a company, can take control of different Subject Instances in different processes. E.g. he can be responsible for the manager subject in the Business Trip Organisation process for his own staff, while for his own business trips he takes on the role the applicant subject with his own superior.

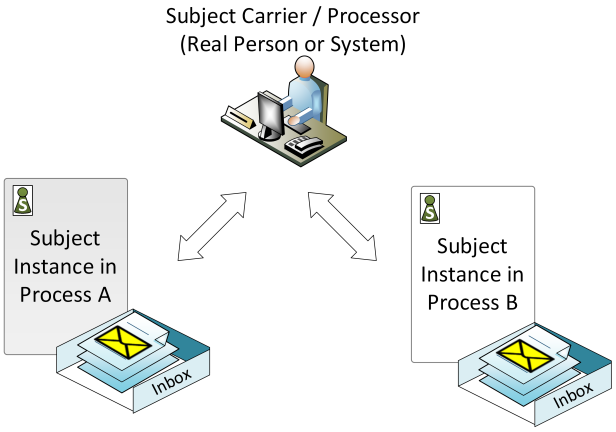


Figure 2.15: Concept of one Subject Carrier(processor) responsible for multiple Subjects in different process, each with its individual inbox([Els19])

Equally, the subject carrier of a single subject could change during the execution of a behavior. The possibility to switch and the according management is part of the functionality of an execution engine and not per-se of the model itself. Currently the PASS modeling standard has no specific mechanism to limit or encourage subject carrier switching.

The Subject vs. Subject Carrier vs. Role vs. Actor vs. Agent

The previous section introduced the concept Subject Carrier as something different from a Subject. Further in the wide field of Computer Science, Business Process Modeling, and IT-System theory, the very similar terms of Actor, Agent, or Role are being used in slightly different but at the same time very similar context to the one here that describes the domain of subject-oriented process modeling with PASS. Often, when learning PASS, people familiar with the mentioned terms wonder how things are related, or if subjects could be called or understand as actors or roles. A full discussion of this topic would require an in-dept discussion of each term and the, supposedly impossible, attempt to find a generally accepted and fitting description for especially the terms of role, actor, and agent.

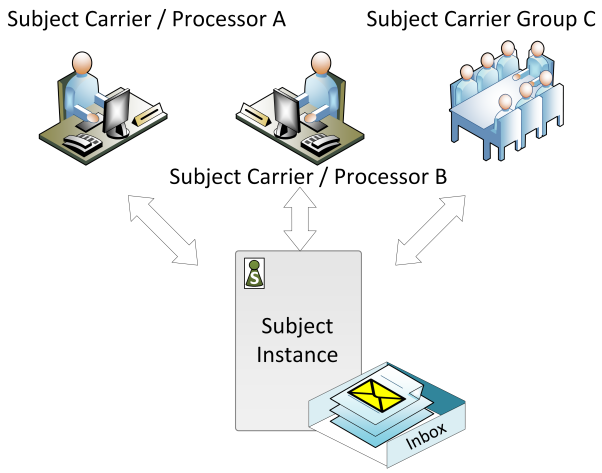


Figure 2.16: Concept of multiple Subject Carriers (processors) that can alternate in the the execution of on subject instance in a process ([Els19])

All of them, including the term of Subject and Subject Carrier, have in common that they refer to or describe something that is an active entity in a specific context. To what degree however differs vastly and depends on the persons understanding for each term.

In the context of PASS, as stated, "Subject" always refers to the element of a process model for whom it is modeled of how it interacts and what behavior it has in a process context. This is very similar to the concept of "role" often used in IT-system management, where for the purpose of giving users certain rights, they often are assigned to certain groups that in turn can take on certain roles. Due to the similarity the concept of role can easily be matched to Subjects in according workflow systems; they may even use the same labels such as 'Manager'. However, the term role has per-se nothing to do with subject-oriented process models and therefor cannot be used synonymously.

The term "actor" can also be used very similar to the term "Subject", and certain modeling SO modeling environments actually prefer it to the term of Subject (Namely the Act'n'Connect modeling tool). It could be argued, that the term actor is more easily understood as an active entity than the term Subject, where subject could also be understood as being a synonym for "topic" instead of referring to the grammatical structure. On the other hand, actor is very generic term and terminology wise makes it hard to differ "actor" as an specific model element from the concept a general active entity e.g. a real-life actor.

An "agent", or "software agent" is an active entity that could also be described as an actor. However, usually a software agent refers to a individual piece of instruction/or code that runs in specific program context. The activities or behavior of software agent is usually described in program code. While a piece of software referred to as an agent could be responsible for executing a Behavior described in a PASS process model, this is only possibility and illustrates how agent and subject are referring to the same concept.

2.6.2 Behavior Execution

As stated before, an SBD consist of States and Transitions used to describe what actions a subject performs in what order. The execution concept is, that a subject always **"is in exactly on state"**. To "be in a state" expresses that the subject is

continuously performing the activity described for that state. The activity of state should be denoted at least by its label, but can also be detailed out with a Function Specification. One or more outgoing transition arrow denotes the next possible state to be executed as well as the condition (**Transition Condition**) that must be fulfilled in order to follow the denoted path.

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 place 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.

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 or followed to the next stat. 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 2.17 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 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.

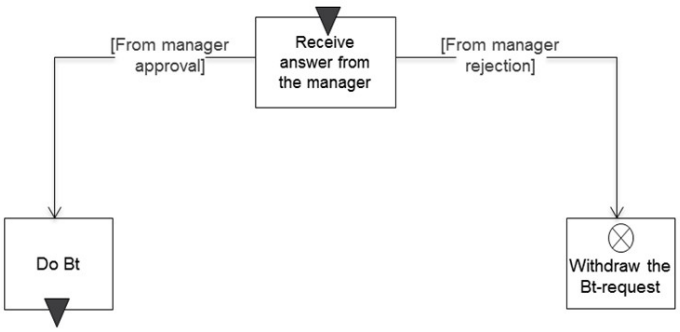


Figure 2.17: Example of alternative receiving

Choice Segment Execution

In principle a subject can be only in one state at once. The Choice segment may seem as somewhat of a contradiction to that statement, however, it is not.

There are two possibility interpretations for the execution of this element, a simple and more complex one.

As an example take a simple Choice segment with three paths, A, B, and C, with A being Mandatory to start and end, B being mandatory to start, and C being optional to start but mandatory to end, as is shown in Figure 2.18.

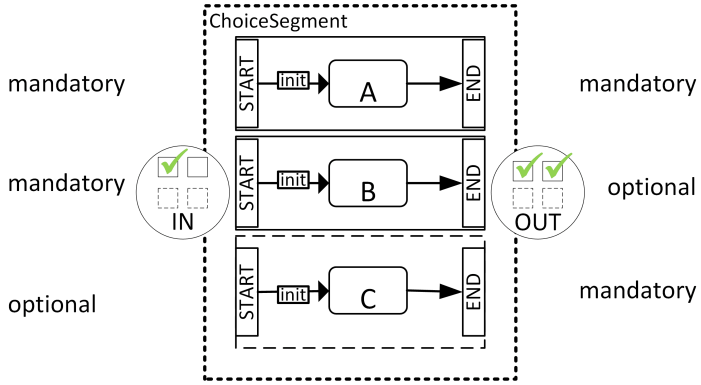


Figure 2.18: Exemplary Choice Segment with three paths, with different sets of mandatory/optional setting

The simple interpretation is to simply interpret the Choice Segment simply as a short cut or a compact variant to modeling a multitude of possible paths as is shown in figure 2.19. Path C can always be circumvented, but once started must be completed. While B cannot be circumvented but can always be "skipped" out once started. A can't be neither. This interpretation works very well especially for one-state paths. If a path contains more states, or even branching, which is technically allowed, this interpretation becomes somewhat restricting, as any path must first be finished individually after starting, before starting the next path segment.

The more complex interpretation of the execution of the Choice Segment does allow switching between the execution of the active state in each separated path upon the arbitrary choice of the subject carrier. However, for that to work, a possible execution engine, in theory, must be able to trace the current state in each path separately and possibly even allow to interrupt the execution

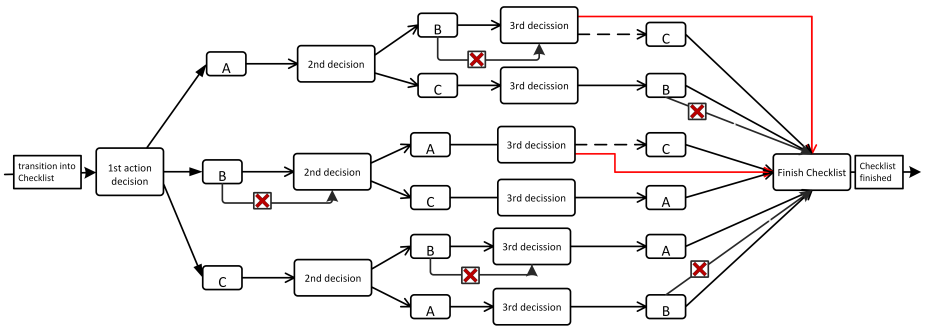


Figure 2.19: Simple Execution Interpretation of a Choice Segment shown in Figure 2.18

of one state temporarily. That would be necessary in order to continue the execution upon the external subject carrier making the choice to switch back to the execution of that path.

Default Functions

DefaultFunctionDo1_EnvironmentChoice DefaultFunctionDo2_AutomaticEvaluation DefaultFunctionReceive1_EnvironmentChoice DefaultFunctionReceive2_AutoReceiveEarliest DefaultFunctionSend

2.6.3 Substitutions for Specialized PASS Transition Execution

In the PASS standard, there are several specialized Transitions. For their execution, they could be interpreted or substituted in the following ways with the standard elements.

Note that all of these are possible technical substitutions that should allow a standard interpreter to handle these transitions. However, an actual workflow execution engine can implement these elements also in other ways.

User Cancel Transitions

,

To interpret and correctly execute, **UserCancelTransition** s, the existence of User or Subject Carrier in an execution software must be assumed. Equally, a technical solution to communicated with this user must be assumed. Consequently, each User-Cancel transition could be substitute as is shown in Figure 2.20. An additional interface subject needs to be added that connects to an implementation that in turn allows interaction with the user beyond the process flow.

Note that this substitution is for User Cancel Transitions originating from Receive States. User Cancel transitions originating from Do States can simply bee seen as relatively specialized Do Transition that allows a user to arbitrarily set the execution of s State Function to be *done*. User Cancels for Send States would need to be substituted similar to a Sending Failed Transition but with the User interacting directly with the Send Controller and deeming a Send activity to be failed.

Sending Failed Transitions

To interpret **Sending Failed Transition** s it is necessary to assume the existence of a technical entity that controls a subject's send activities during execution; the

Send Controller. After have been given the command to forward a message to a specific subject instance, the send controller does inform a Subject's Behavior Interpreter about a the possible success or failure (after a timeout) and the subject reacts accordingly as depicted in Figure 2.21.

Timer Transitions

There are the two general types of time based transitions:

Substitution **Timer Transition** s and **Reminder Transition** . Both work a little different from each other and also different when originating from each of the different state types.

If the execution envoironment is able to handle it, **Timer Transition** , also known as *Time-Outs*, can be executed by adding an accordingt time-out funcionallity to a State's internal function.

Otherwise, certain substitutions could be done in order to realizer the execution of Time Transitions. All of them have in common that they require the assumed existence of a timer or calendar subjects that theoretical handles all time calculations and sends an according trigger message when the time has come.

Timer from Receiver States

The substitution done for Timer Transitions originating from Receive States. Here, before entering a state with an outgoing timer transition, the according time-out time from that transition needs to be set/send to the timer and the timer transition itself can be substituted simply by another receive transition from that state. If the state can be visited multiple times, the timer will also need to be deactivated. Otherwise, the according time-out message might still be sent and in the subject's inbox when reentering the state.

Timer from Send States

Instead of setting a time with an assumed timer subject, a time-out transition originating from a send state is equal to a Sending Failed Transition with a modified time-out-time that is different from an execution environments default value. The Send Controller (see Section 2.6.3/ Figure 2.21) simply needs to be informed about that specialized time-out time and send replies accordingly.

Timer from Do States

Realizing time-outs for Do States is more complex. It could be done via a modification to that states internal Function Specification to include an additional internal time-out concept and substituting the outgoing Timer Transition with normal Do Transition that has the Transition Condition of "time is up".

A more complex substitution is shown in Figure 2.24. Here, the concept of a Guard or Interrupt Behavior is employed, that guards only the single state that had the original time-out transition going out of it. The guard is triggered by a time-out message from a timer subject that needs to be set before entering the guarded state.

Reminder Transitions

Reminder Transitions are different from time-outs, as they are not triggered upon the passing of a specific amount of time after entering the state but rather, in terms of time since the transition has been traveled last or upon the passing of an calendar time interval.

Naturally, they are used only in States that will be (re-)visited multiple times and only **Receive States** and **Do States** are valid source states for these transitions.

Substitution is similar to that of Timer Transitions. Again a Timer or Calendar Subject is assumed to exist, that is able to handle time based event calculation and send the according reminder messages upon reaching the time based conditions. Also, that timer needs to be set before entering the original source state for the first time. In contrast to a timer, the reminder message is send immediately so that transition could be traversed. In order to be able to send the next reminder correctly (be it 4 weeks or 4 seconds) the time must be informed if said transition has been followed as is shown in Figure 2.25.

To keep the the number of reminder messages under Control an additional **Input Pool Restrictions** for that limits the number of reminder messages from that Timer Subject to one is useful, together with the **Input Pool Constraint Handling Strategy [of] Delete Oldest** for that message.

2.7 Variants of PASS Visualizations

Even while being a formal language, the Parallel Activity Specification Schema (PASS) is always also a graphical process notation. Throughout this chapter, a generic visualization was used.

However there are several modeling programs or tools in existence intended for modeling PASS (See [FBE+17]). While all of the tools more or less adhere to this standard, many tools use slight variations in their graphics to visualize PASS. This section shows a selection of various PASS styles.

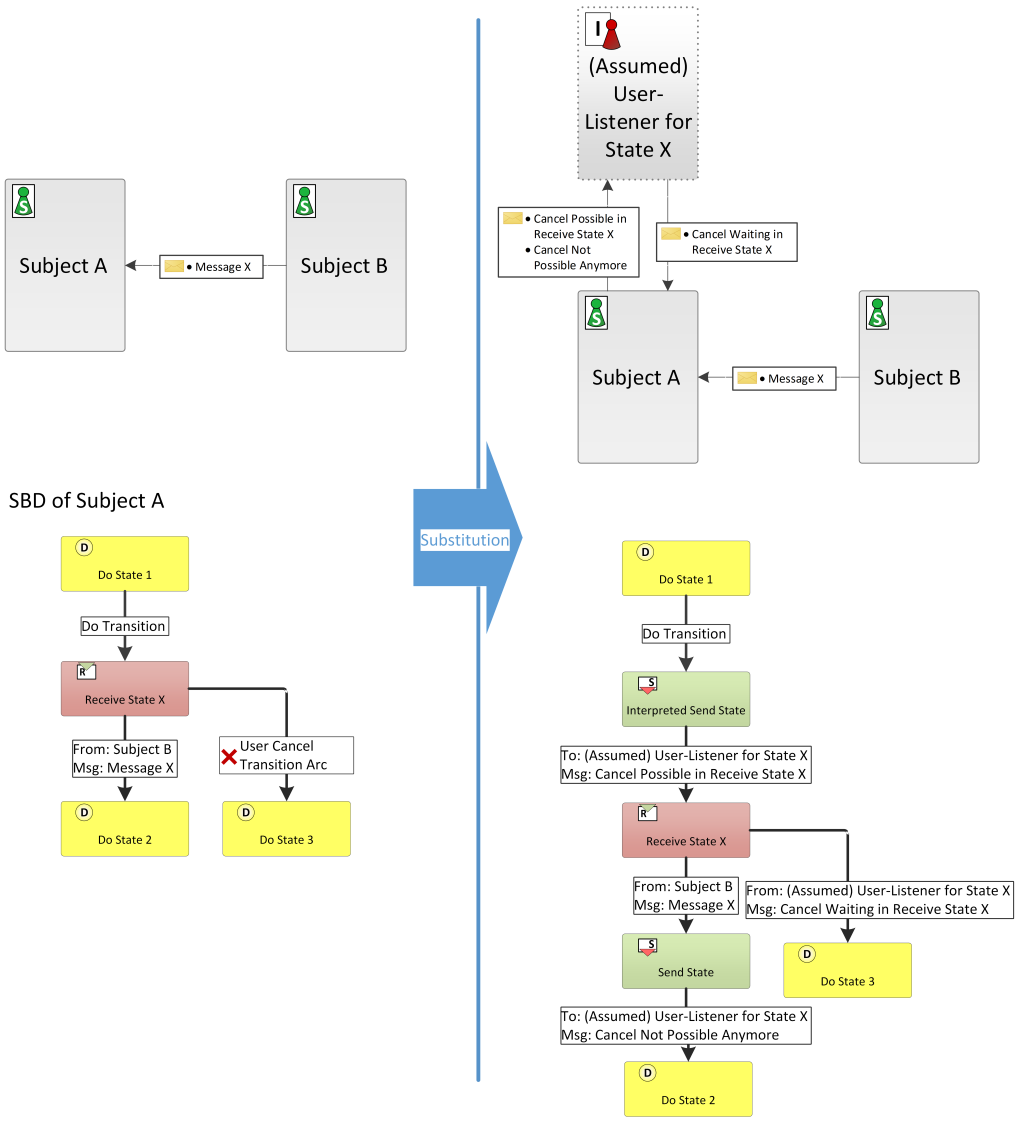


Figure 2.20: Possible Substitution of a **UserCancelTransition** for execution

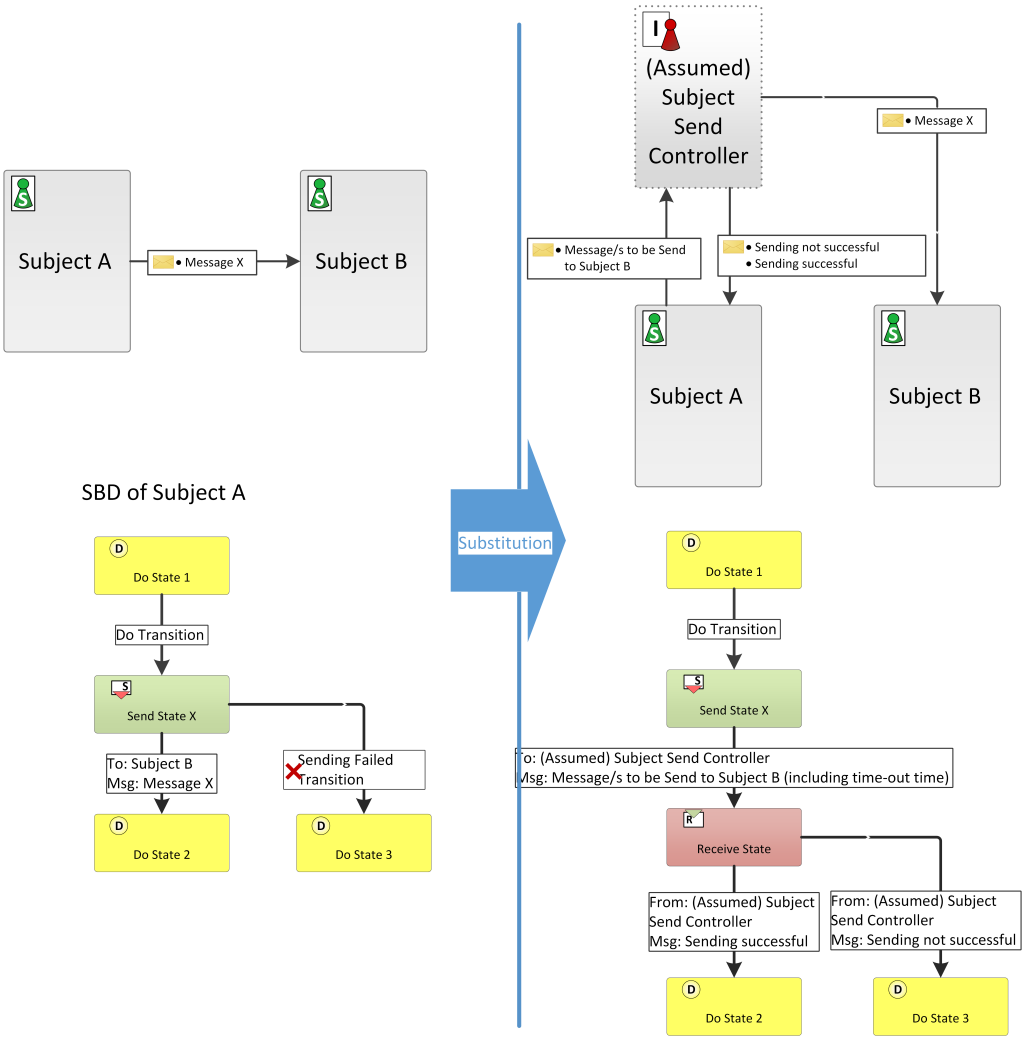


Figure 2.21: Possible Substitution of a **SendingFailedTransition** for execution

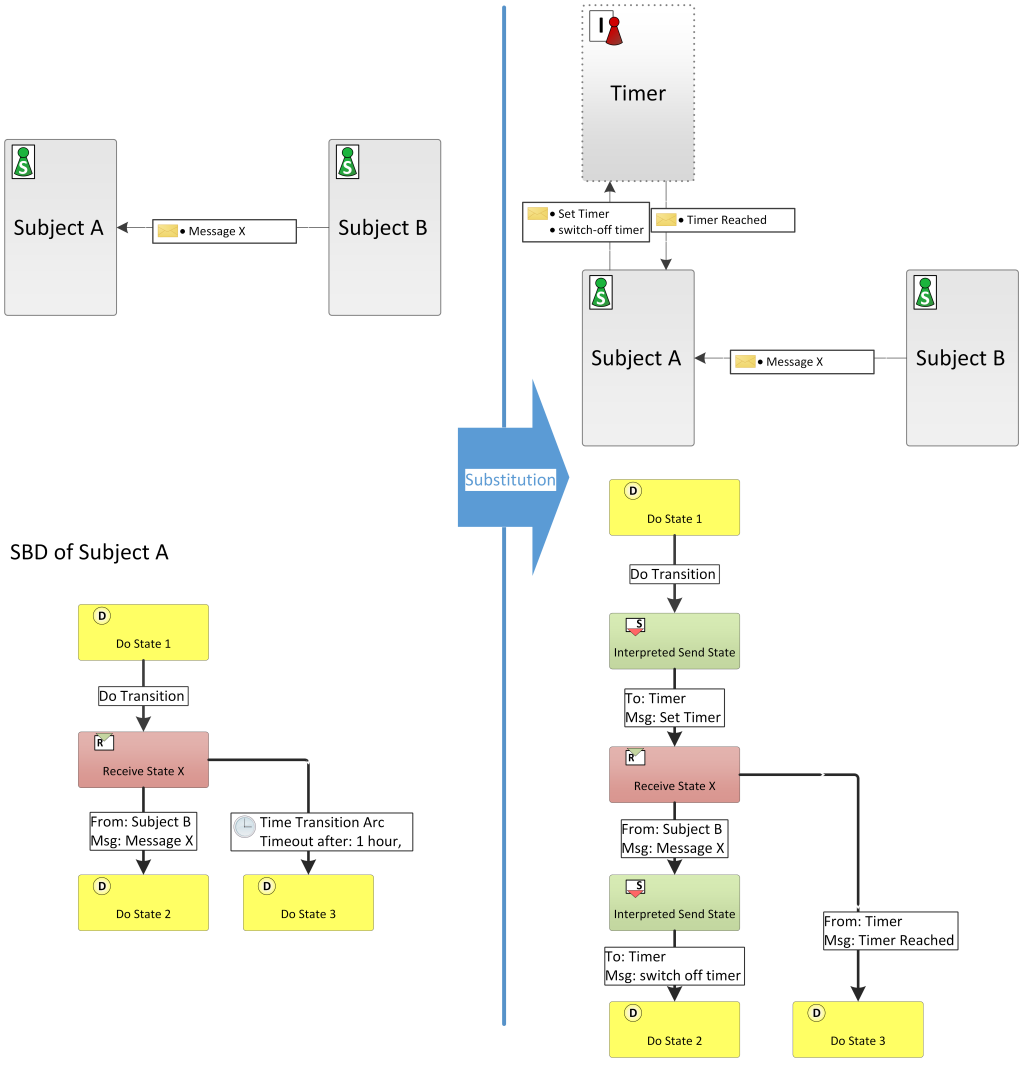


Figure 2.22: Possible Substitution of **TimeOutTransiton** s from **Receive State** s for execution

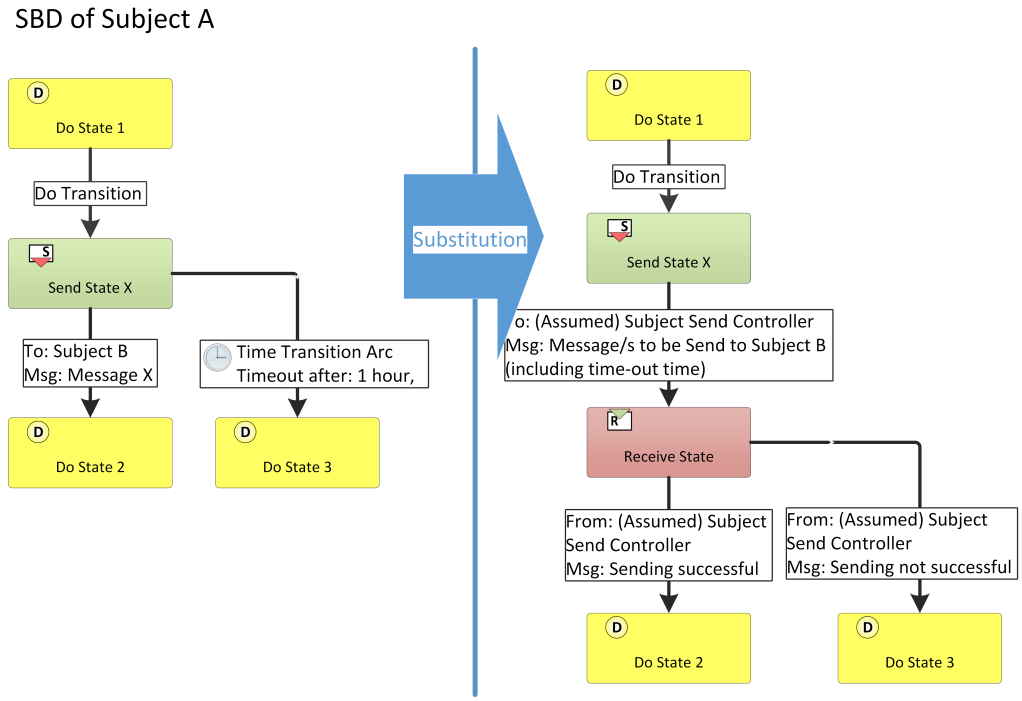


Figure 2.23: Possible Substitution of TimeOutTransiton s from Send State s for execution

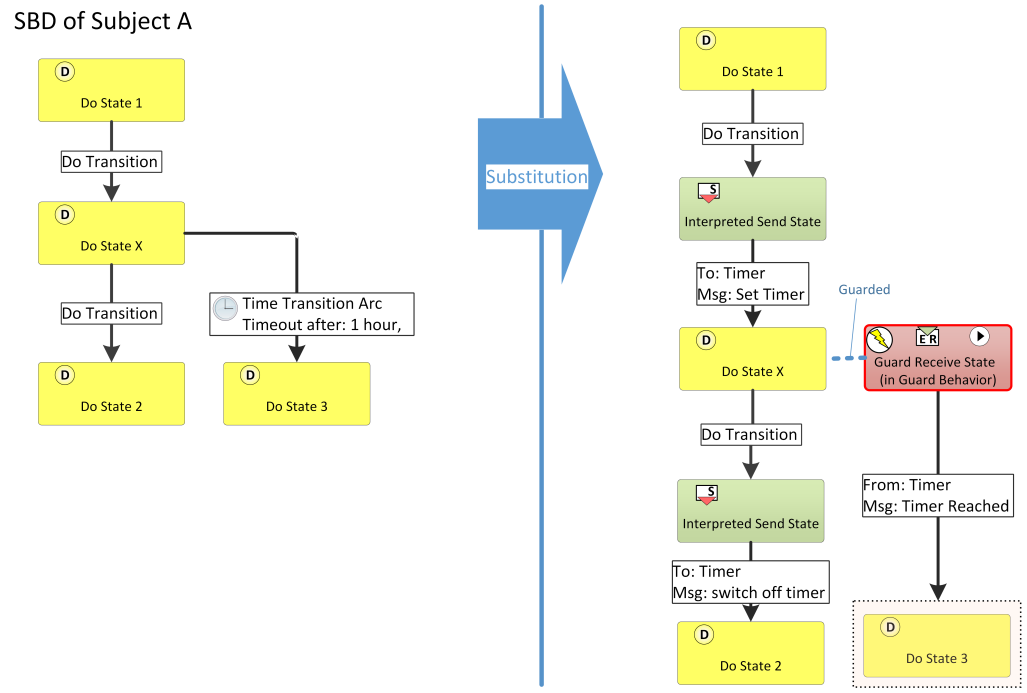


Figure 2.24: Possible Substitution of TimeOutTransiton s from Do State s for execution

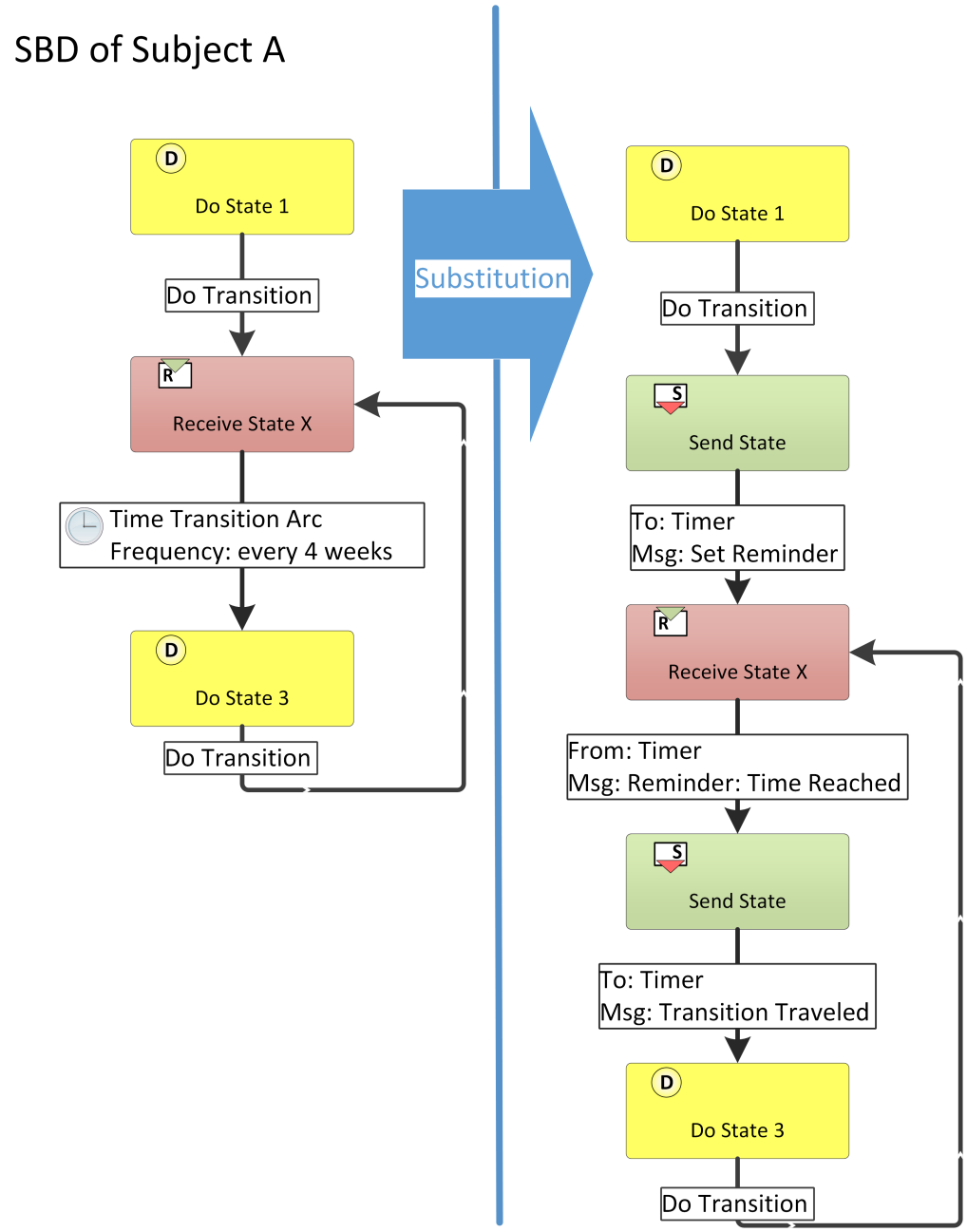


Figure 2.25: Possible Substitution of **Reminder Transition** s from **Receive State** s for execution

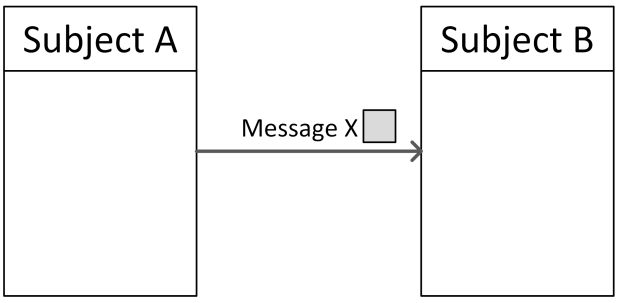


Figure 2.26: Generic Standard PASS SID Symbols

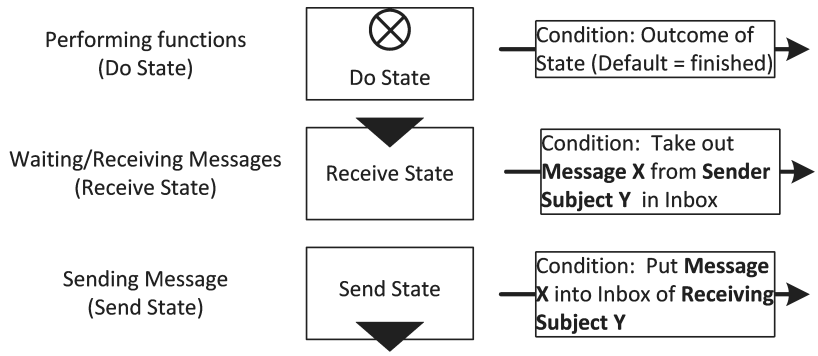


Figure 2.27: Generic Standard PASS SBD Symbols

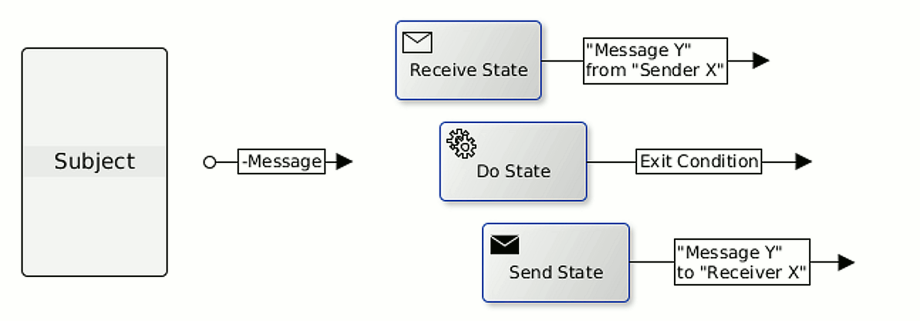


Figure 2.28: Standard PASS Symbols Darmstadt Modeling Engine

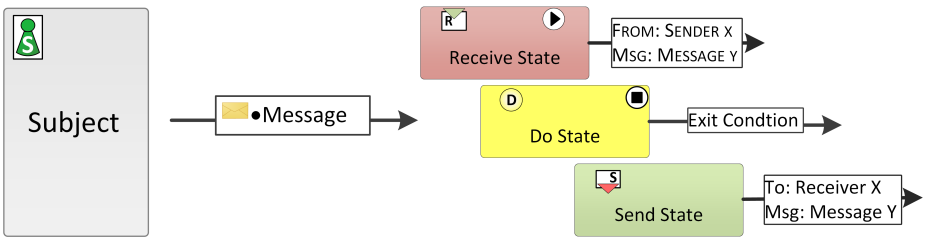


Figure 2.29: Standard PASS Symbols Visio Tool

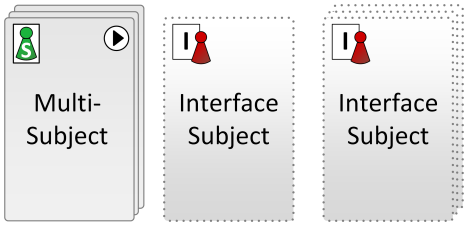


Figure 2.30: Extended SID Elements (Visio Tool)

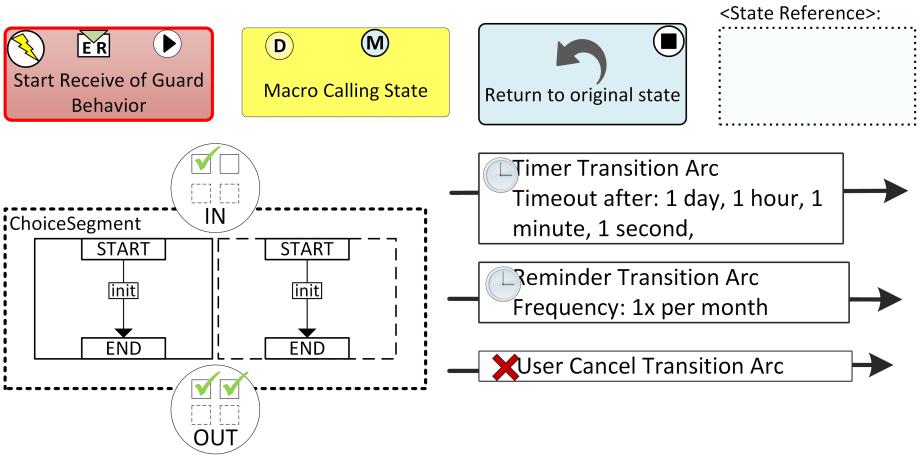


Figure 2.31: Extended SBD Elements (Visio Tool)

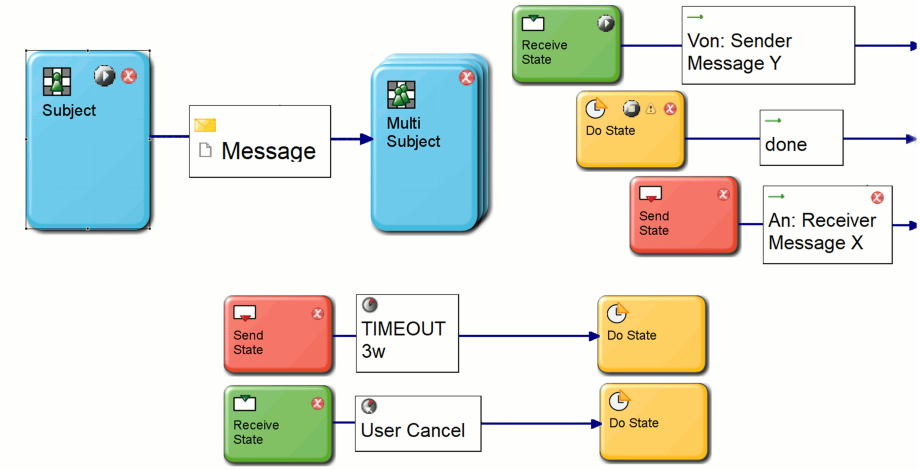


Figure 2.32: PASS Elements in MetaSonic Build tool

CHAPTER 3

Formal Definition of PASS

The previous chapters introduced the general concept of Subject-Oriented process modeling with the Parallel Activity Specification Schema and the corresponding execution concept. This basically does the same, however in a formal manner.

Since description of passive model structure and defining of active execution concepts is different, different formalism are used for each.

3.1 Formal Definition of PASS Process Models in OWL

As mentioned in previous sections, the formal specification for the structure of the Parallel Activity Specification Schema process modeling language exists as an ontology formulated in the Web Ontology Language (OWL). This ontology is called the **Standard_PASS_Ont** and currently hosted at: `https://github.com/I2PM/Standard-Documents-for-Subject-Orientation/blob/master/PASS-OWL/standard_PASS_ont_v_1.0.0.owl`.

In OWL there four principle concepts and color coded as shown here: **Classes**, relationships between classes, the so-called **Object Properties**, non-linking **Data Properties** that are attached to classes and allow attaching individual data values to instances of the classes, and finally the instances of classes themselves, the OWL **Individuals**.

3.1.1 Application Concept of the Standard PASS Ontology

The usage of OWL as the technology for the specification comes with an intended concept of how the Standard PASS Ontology is supposed to be used as the technical foundation for an model exchange standard, next to its being a formal definition of PASS on a conceptual level.

In principle, the structure of PASS is defined with **Classes**, **Object Properties**, and **Data Properties** in the Standard PASS Ont. The actual PASS process models consist of OWL **Individuals** that adhering to the classes and rules defined in the Standard PASS ont, but will be saved in their own individual OWL Files or equivalent OWL/RDF storing systems.

The following Figures 3.1 and 3.2 taken from [Els17] describes the application concept for how a process model should be saved in an individual model ontology file while referring to the standard definition ontology. At the same

time, an according model may refer to other ontological specifications that contain additional but logically consistent and compatible, extensions to the standard.

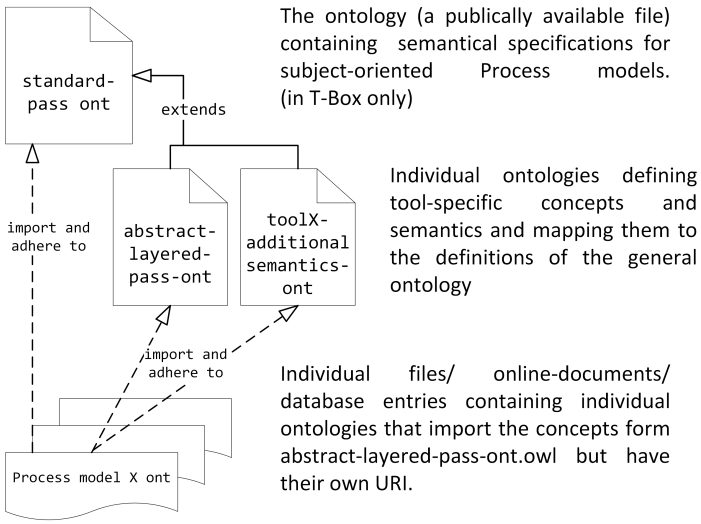


Figure 3.1: Proposed use and interaction of ontologies (from [Els17]).

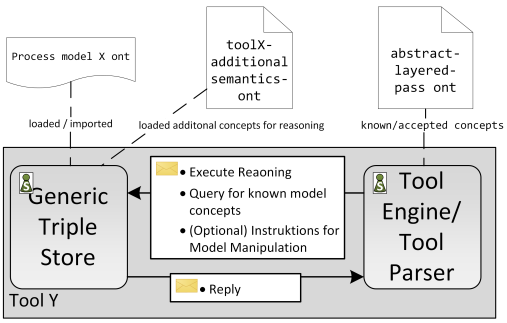


Figure 3.2: Conceptual Import Process (SID) (from [Els17]).

3.1.2 The Principle Structure of PASS in the Standard

As a formal specification document, the Standard PASS Ont currently is comprised of roughly 4000 lines of XML/RDF/OWL code[1]. Ideally, it should be perceived using a specialized ontology/OWL editor such as e.g. the open source tool Protegée.

Listing the entire content and detail of the specification here in written form is not useful; introducing the general structure as orientation, however is.

The following figure 3.3 gives an overview of the general structure of the PASS OWL specifications.

All elements of a PASS model are instances of sub classes of the general super-class **PASSProcessModelElement** . **PASSProcessModelElement** has five sub-classes (subclass relations 084, 055, 069, 001 and 085 in figure 3.3) that together form PASS process Descriptions. Instances of the classes **PASSProcessModel** , **DataDescriptionComponent** , **InteractionDescribingComponent** are

[1]This may differ in other OWL encoding styles such as Turtle script

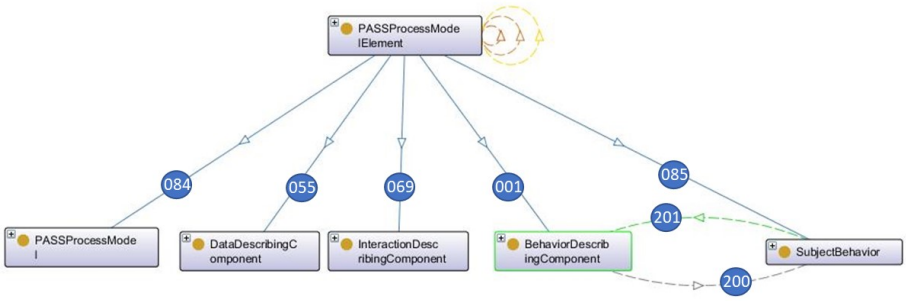


Figure 3.3: Elements of PASS Process Models

used for defining the structural aspects of a process specification in PASS. The classes **SubjectBehavior** and its content element the **BehaviorDescribingComponent** define the dynamic aspects, namely in which sequences messages are sent and received or internal actions are executed.

3.1.3 PASS Process Model Elements and PASS Process Model

Any **PASSProcessModelElement** , including the **PASSProcessModell** itself has at least one name (**hasModelComponentLabel**) and exactly one individual ID (**hasModelComponentID**)[2].Any model element also can be equipped with **Additional Attributes** (**hasAdditionalAttribute** - property 208 in 3.3).

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 3.4 shows the corresponding ontology for the PASS process models.

The class **Subject** and the class **MessageExchange** have the general relation (**hasRelation**) to any other `model elements` as well as the class **PASSProcessModel** (property 226 in 3.3).

The properties **hasReceiver** and **hasSender** express that a Message Exchange has a sending and receiving subject (properties 225 and 227 in 3.3) whereas the according inverse properties **hasOutgoingMessageExchange** and **hasIncomingMessageExchange** define which messages are sent or received by a subject. The property **hasStartSubject** (property 229 in 3.3) defines a start subject for a **PASSProcessModell** . A start subject is a subclass of the class subject (subclass relation 122 in 3.3).

3.1.4 Interaction Describing Components

The following figure 3.5 shows the subset of the classes and properties required for describing the interaction of subjects, the **Interaction Describing Components** .

The central classes are **Subject** , **MessageSpecification** , and **MessageExchange** . Between these classes are defined the properties **hasIncomingMessageExchange** (in figure 3.5 number 217) and **hasOutgoingMessageExchange** (in figure 3.5 number 224). These properties define that subjects have incoming and outgoing messages. Each **MessageExchange** has a sender and a receiver (in figure 3.5 number 227 and number 225). Messages Exchanges also have a type. This is expressed by the property **hasMessageType** (in figure 3.5 number

[2]When OWL technology is employed the ID of a model element ideally coincides with its URI or parts of it. However, not necessarily

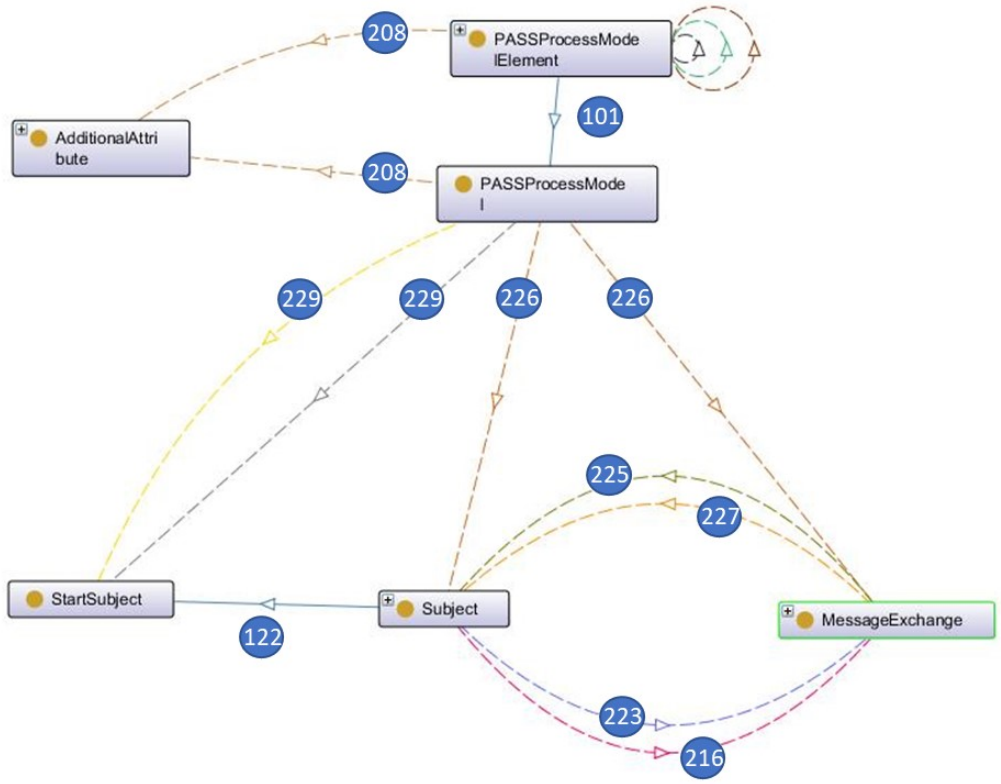


Figure 3.4: PASS Process Modell

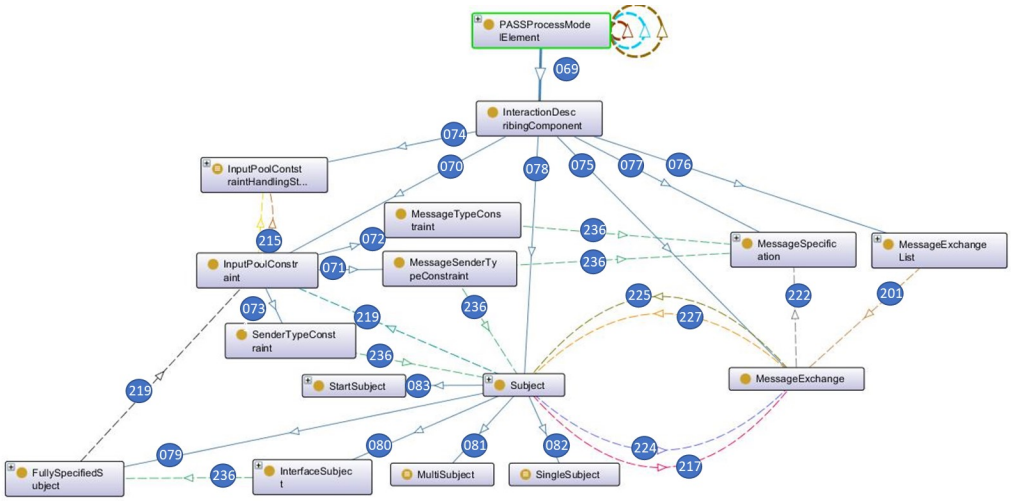


Figure 3.5: Subject Interaction Diagram

222) linking it to a **MessageSpecification** . These Message Specifications are the actual existential definitions for Messages, while the model element of the Message Exchange are used to define that an existing message is indeed exchanged between specific subjects. Beyond that, message exchanges that have the same sender and receiver may be grouped into an **MessageExchangeList** that **contains** them.

During execution, each Subject has an Input Pool. Input pools can be restricted by three types of constraints (see section 2.2.5). This is expressed by the according property references (in figure 3.5 number 236) and **InputPoolConstraints** (in figure 3.5 number 219). Constraints which are related to certain messages have references to the class **MessageSpecification** .

There are four sub-classes of the class **Subject** (in figure 3.5 number 079, 080, 081 and 082). Where the **FullySpecifiedSubject** and the **InterfaceSubject** correspond directly to their concepts as described in Chapter 2.2, the other threee entries (**StartSubject** , **SingleSubject** , and **MultiSubject**) are indirect definitions.

A **StartSubject** (in figure 3.5 number 83), is any *FullySpecifiedSubject* with a behavior that starts in **Do State** or **Send State** . In other words, a subject that is not started by other subjects in a process context. In certain execution context only one Start Subject may be allowed per process to be considered valid. However, the standard for PASS in principle does allow multiple Start Subjects per process.

Single Subject s and **Multi Subject** s are *equivalent classes*. A **Single Subject** is equivalent to any Subject that **has [a] Maximum Subject Instance Restriction** of **one**, while a **Multi Subject** is equivalent to any Subject that **has [a] Maximum Subject Instance Restriction** **larger than 1**.

3.1.5 Ontology Structure for Behavior Description

Each **Fully Specified Subject** contains at least one **Subject Behavior** (**containsBehavior**), which is consider to be its **Base Behavior** (see property 202 in 3.6 — **containsBaseBehavior**) and may have additional subject behaviors (see subclasses of **SubjectBehavior** in 3.6) for **Macro Behavior** s and **Guard Behavior** s.

The details of all behaviors are defined as state transition diagrams (PASS behavior diagrams). These Behavior Diagrams themselves **contain** **Behavior Describing Component** s (see figure 3.6). Inversely, the **Behavior Describing Component** have the relation **belongsTo** linking them to one **Subject Behavior** .

Behavior Describing Components

The following figure shows the details of the class **BehaviorDescribingComponent** . This class has the important sub-classes **State** , **Transition** and **TransitionCondition** (see figure 3.7). The sub-classes of the state represent the various types of states (class relations 025, 014 and 024 in 3.7). The standard states **DoStates** , **SendState** and **ReceiveState** are sub-classes of the class **StandardPASSState** (class relations 114, 115 and 116 in 3.7). The subclass relations 104 and 020 allow that there exists a start state (class **InitialStatOfBehavior** in 3.7) and none or several end states (see subclass relation 020 in figure3.7) 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 3.7.

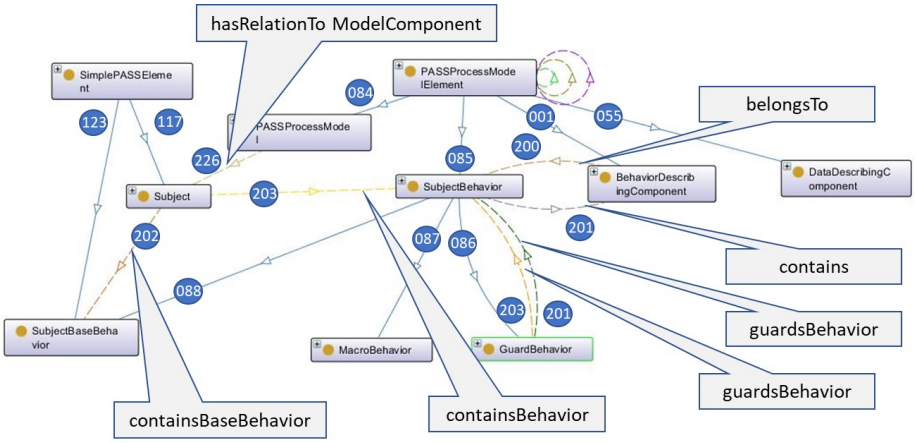


Figure 3.6: Structure of Subject Behavior Specification

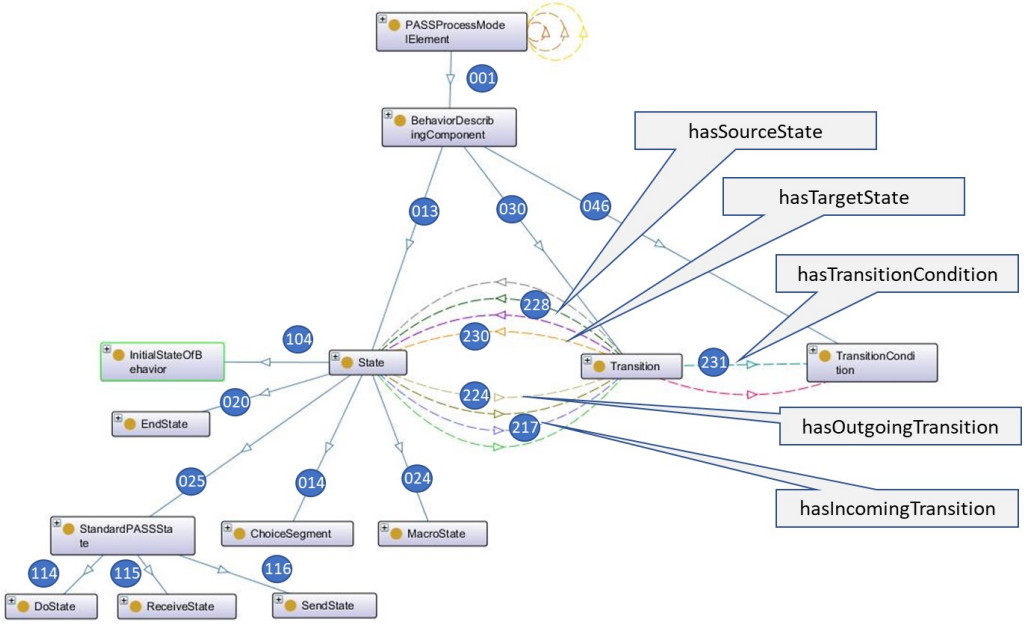


Figure 3.7: Subject Behavior describingComponent

States can be starting and/or endpoints of transitions (see properties 228 and 230 in figure 3.7). This means a state may have outgoing and/or incoming transitions (**hasIncomingTransition** and **hasOutgoingTransition** - see properties 224 and 217 in figure 3.7). 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 3.8 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).

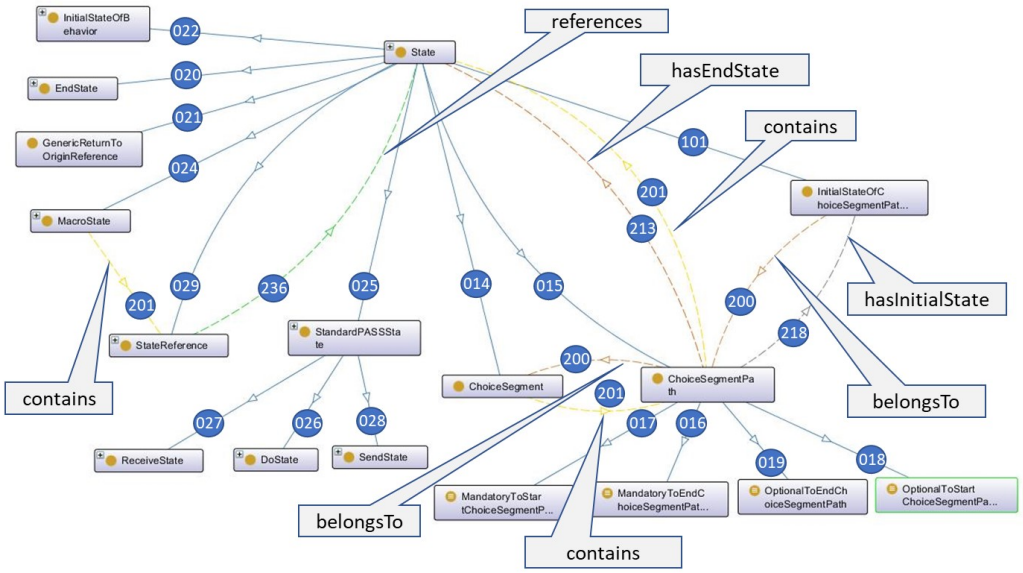


Figure 3.8: Details of States

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 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 3.7). 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 3.9). The class **CommunicationTransition** is divided into the sub-classes **ReceiveTransition** and **SendTransition** (sub-classes 32 and 33 in figure 3.9).

Each transition has depending from its type a corresponding **Transition Condition** (property 231 in figure 3.9) which defines a condition that must be valid in order to execute/travel a transition. This can be

Next to these main PASS transitions, the following specialized transitions exist **UserCancelTransition** s, **SendingFailedTransition** s, and **TimeTransition** s.

3.1.6 Data Describing Components

Each subject encapsulates data (business objects). The values of these data elements can be transferred to other subjects. The following figure 3.10 shows the structure of the model components concerned with data structure description and data handling.

Three subclasses are derived from the class **DataDescribingComponent** (in figure 3.10 are these the relations 060, 056 and 066). The subclass **PayLoadDescription** defines the data transported by messages. The relation of **PayloadDescriptions** to **MessageSpecification** is defined by the property **containsPayloadDescription** (in figure 3.10 number 204).

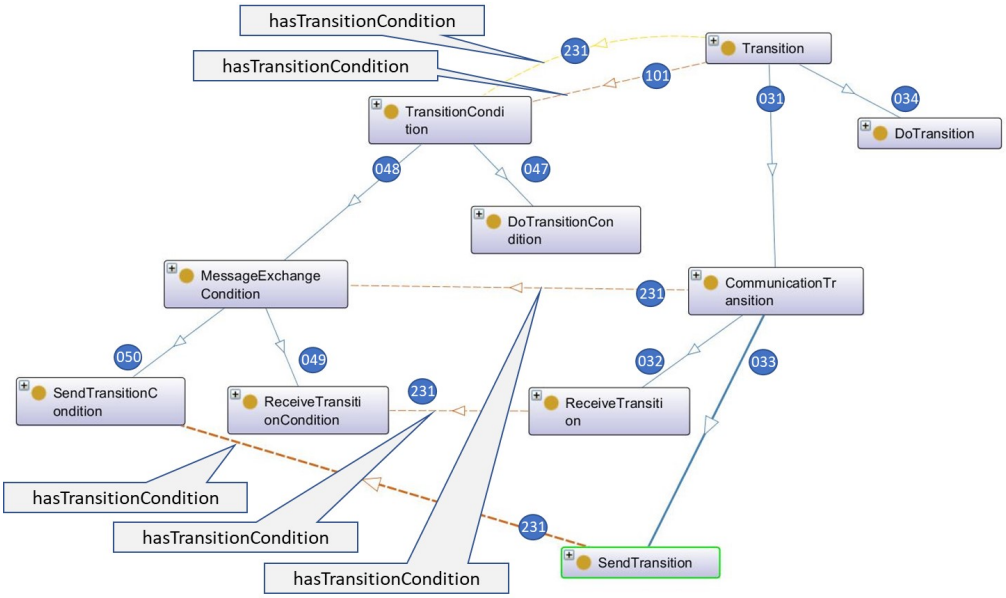


Figure 3.9: Details of transitions

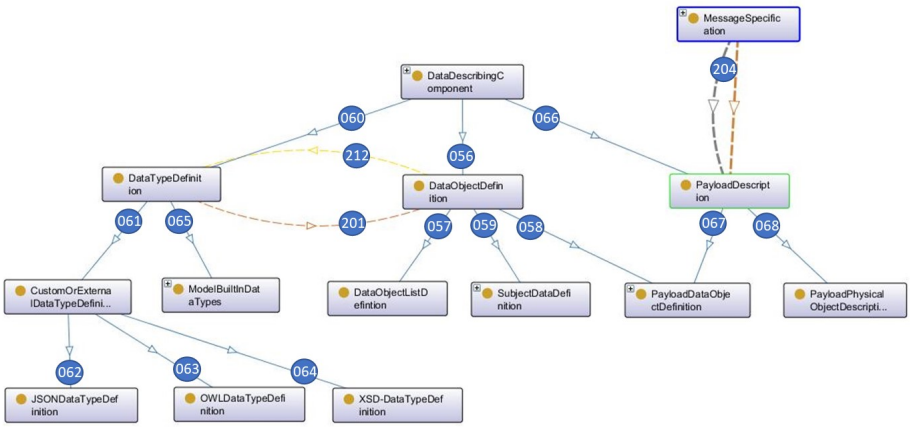


Figure 3.10: Data Description Component

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 in models that describe such concepts. The class **PayLoadDataObjectDefinition** is used to descripe data payloads (Subclass relations 068 and 67 in figure 3.10). These payload objects are also a subclass of the class **DataObjectDefinition** (Subclass relation 058 in figure 3.10).

Data objects have a certain type. Therefore class **DataObjectDefinition** has the relation **hasDatatype** to class **DataTypeDefinition** (property 212 in figure3.10). Class **DataTypeDefinition** has two subclasses (subclass relations 061 and 065 in figure 3.10). 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 3.10).

3.2 Formal ASM Definition of Subject Execution

3.2.1 Introduction

While the the correct structure and syntax of Parallel Activity Specification Schema (PASS) Process Models are specified in the Web Ontology Language (OWL), its execution semantics are defined using the formalism of Abstract State Machines ASM as defined by Börger in [BS03] and introduced in Section 1.3.

The original PASS ASM execution specification was formulated in [Bör11] in 2011, but is now somewhat outdated in terms of PASS model elements covered in the specification as well as the used vocabulary.

While not yet covering all PASS elements, the current version of PASS execution is based on [WBH19b] and [WBH19a]. This version is not a pure ASM specification, but rather a specification meant for *CoreASM*, an open-source ASM execution engine/software implemented for the JAVA Virtual Machine (JVM)[FG11]. Thereby, it is basically an execution specification as well as a reference implementation at the same time[3].

This section provides a commented overview over the CoreASM PASS Reference Implementation. Only a reduced set of language elements are shown. Advanced/sub functions and implementation details are left out, which allows 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. The complete spec is provided in Appendix C.

3.2.2 Differences

Note that there are some conceptional differences between the ASM reference implementation and the OWL description. Important differences are:

- The End State concept is not supported, but instead a *Terminate Function* is used.
- 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*, which are not part of the OWL-Structure Standard Definition.
- 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 comparison of the conceptual differences is given in Appendix Section C.2.

A comparison that systematically matches the ASM Execution concepts with the the OWL-Model concepts is given in Appendix Section A. Furthermore, a concept for advanced execution semantic can be found in Appendix 5.8

[3]Next actual ASM spec and the CoreASM framework, the referential implementation is also complemented by 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 Appendix Section C.1.

3.2.3 Foundation

The specification makes use of concepts 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 *process instance* has a unique *ProcessInstanceID* (short: `PI`) assigned. Within a process instance multiple instances of a subject (*Subject ID*) can exist (Multi Subject concept). Each subject instance has an *agent* assigned, in order to distinguish subject instances and to keep track of its current "*state*"[4]. The agent is identified by its 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`). The concept of Channel itsef is used in the *"mobility of channels concept"* in order to support distributed communication patterns. The overall subject *"state"* is stored in the following 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 the triple of the Subject's Channel (a list), the scoped Macro Instance ID (a Number) and the Variable's name (a String) to a pair of the data type and value (another List). 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 general 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.

[4]State is in the sense of which subject data is present, the content of the input pool queues and also the active SBD states

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

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

Listing 2: variable

Messages

A Message 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.

3.2.4 Interaction Definitions

The interaction between Subject Instances is implemented as asynchronous message exchange. Messages are placed into the *Inputpool* of the receiver where they are retrieved from when the receiver is in a Receive State.

For the communication with MultiSubjects, i.e. sending the same message to multiple instances/agents of one Subject, a *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 when all reservations can be placed, they are afterwards replaced with the actual message.

Inputpool

The Inputpool of a subject instance is structured into multiple FIFO queues, distinguished by Subject ID of the sender, the Messagetype and the CorrelationID. This way all Inputpool Restriction Types can be realized, be it by general maximum number of Messages or discrimination according to sender Subject ID and Messagetype; in case the Inputpool limit is reached no additional reservation can be placed. Similarily, in order to realize the proper termination of a Subject Instance, a specific queue (and also the complete Inputpool) can be *"closed"*, in which case also no additional reservation can be placed into the input pool in general.

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

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

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

Listing 3: inputPool

The queues of the Inputpool are stored in the `inputPool` function. The function `inputPoolDefined` is used to keep track of the locations of the queues that

are in use, so that their content can be checked upon termination. The function `inputPoolClosed` is used to store whether a queue is closed due to restrictions. 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 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.

3.2.5 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 aborted the `Perform` rule calls the state behavior of the underlying function until it is `completed`.

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

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

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

Listing 6: Behavior

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

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

Listing 7: wantInput

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

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

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

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

Listing 8: SelectTransition

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

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

3.2.6 Internal Action

The *Internal Action* is used to model the execution of 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.

3.2.7 Send Function

Straight forward, the Send Function sends a Message. Reminder: Disregarding the optional Timeout and Cancel transitions, a Send State in the model 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 instance 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.

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

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

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

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

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

Listing 12: StartSend

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

Listing 13: PerformSend

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

Listing 14: SetMessageContent

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

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

Listing 15: DoReservations

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

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

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

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

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

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

Listing 16: DoReservation

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

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

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

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

3.2.8 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

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

    SetCompleted(MI, s)
  }
```

Listing 17: TryCompletePerformSend

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

      EnsureRunning(r)
    }

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

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

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

Listing 18: PerformTransitionSend

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

Listing 19: ReplaceReservation

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

    SetAbortionCompleted(MI, s)
  }
```

Listing 20: AbortSend

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

Listing 21: CancelReservation

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

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

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

Listing 22: PerformReceive

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

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

Listing 23: CheckIP

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

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

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

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

Listing 24: PerformTransitionReceive

The transition function removes the available Messages from the 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.

3.2.9 Modal Split and Modal Join Functions

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

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

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

Listing 25: ModalSplit

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

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

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

Listing 26: ModalJoin

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

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

3.2.10 CallMacro Function

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

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

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

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

Listing 27: macroTerminationResult

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

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

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

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

The second phase evaluates the created Macro Instance.

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

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

3.2.11 Terminate Function

The Terminate 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 Terminate Function terminates the Subject.

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

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

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

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

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

Listing 28: CallMacro

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

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

Listing 29: InitializeMacroArguments

```
rule PerformTerminate(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: PerformTerminate

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, which will block future subject starts, if there are remaining messages left in the Input-pool, which means, that the subject is *non-proper* terminated.

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

CHAPTER 4

Implementation of Subject-Oriented Models

4.1 "Implementing" a Process Model for Execution

Subject-oriented process models describe the structure as well as the internal aspects of a process system. They can cover organizational as well as technical aspects.

Most finished process models are intended to be used as instructions for some kind of process executing system. Such a system can be completely technical consisting only of (data processing) machines with software components. Alternatively, it can consist of human beings. Or, most likely, it can be a mixture of both — a so called a socio-technical system. All entities that can execute process tasks, be hit human beings, be it production machines, or be it software components are referred to as *"agents"* that constitute the process execution system.

The activity of "making" any system execute a process is called *implementing the process*. Necessarily, this very likely involves a process model as written artifact that contains the instructions about how a process is supposed to run, and when and how.

There usually is a correlation, between the process model and the execution system due to model often being made to fit into the execution system. However, both exist independently. This is also true for subject-oriented process models, even though Subjects are often abstractions of parts of the execution system's agents or at least based on them and the Subjects' interaction often resemble parts of the execution system's structure, but they are not the same! Seen from the other side, PASS model makes implications and contain implicit assumptions about the executing system, but they are only abstraction of it and a real executing component — a potential subject carrier — may be responsible for the execution of various different subjects.

A person responsible for implementing a process therefore needs to understand both, the process system and the execution system and match them accordingly. For subjects to be executed by humans this means simply providing the "correct" people with the "correct instructions". For subjects that are exe-

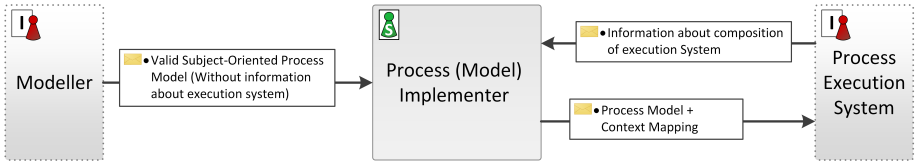


Figure 4.1: SID of the Implementation Process

cuted with the help of or directly by machines and software this means that the according IT system must be programmed or configured "correctly"[1].

In both cases and in order to implement a process successfully, it is important that the Implementer understands the modeled process as well as intended system sufficiently good.

For modeling and understanding the process itself, PASS should be used as intended. However, understanding the system/organization and its internal structure is a different problem. Socio-technical systems usually are complex and there are various modeling approaches that are intended for describing and analyzing an organizational structure or architecture.

4.1.1 Mappings

As mentioned before, on an "atomic" level any socio-technical system consist of various "types of agents", individual human beings, physical machines that are meant for single tasks (e.g. production machines for stamping), and IT components that are used to coordinated them. When implementing a subject-oriented process model, it is necessary to state which "agent" is responsible for the execution of which subject. This concept is called *mapping*; as a verb *"to map"* subjects to agent, or as noun to create *"a mapping"* between subjects and agents.

To be useful, these mappings must be written in a form that can be understood by the execution system in order to assign the correct agents to be subject-carriers for the correct subjects. The mapping is used to answer question like: "Is an agents allowed to execute a subject?", "What resources/agents are available to execute a subject?".

In principle, mappings occurs for both, the social/human and the technical aspects of an execution system and may be done formal as well as informal. However, in praxis mappings are of most importance for technical (IT) workflow execution systems that needs to give tasks to its human users or start automation task and allocate computing resources.

Also very important to understand is that this mapping activity must be (re-)done any time any side of the mapping changes! This is the case for a new process model, a process model becoming obsolete or updated (e.g. with new subjects), or if the available agents in the execution system changes — which can be quite frequently. Furthermore, temporary mappings that are only valid for certain periods of time are also possible.

Simple Static Mapping

A mapping between subjects and agents can be done simple and direct (or "static") as depicted in Figure 4.2. Here, agents are directly mapped to subjects and therefore eligible to execute them. The only information necessary to know

[1]The question about what "correct" is in what circumstances,or what is not correct, is in itself a completely different and complex question.

about the execution system is the existence of the agents in order to assign subjects to them.

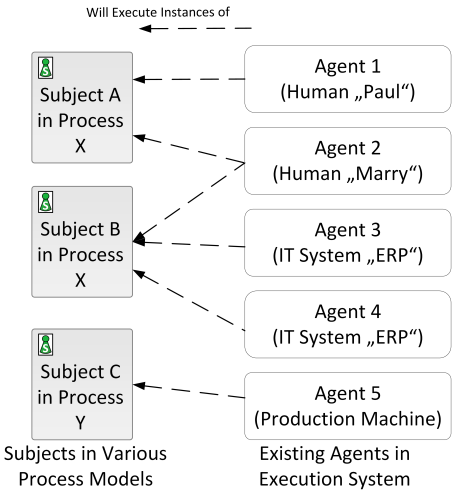


Figure 4.2: Simple Static Mapping of Subjects and Execution System Agents

Complex/Indirect Static (Type) Mapping

In situations where many process models and therefore subjects need to be mapped to a large number of possible agents, mapping would be required in depth quite often. Therefore it is a good idea to do the mapping indirectly.

Even outside the context of process model mappings, the agents of an execution systems are usually grouped or classified. This kind of abstraction over agents is called different things. It can referred to as creating *Agent Types*, it can be called having *Roles* that agents may fill, or general (abstract) org units.

This way, on both sides concepts are mapped to this abstract types. New, or changed subjects are assigned to be executable by certain types/roles while new agents/users are assigned to be able to fulfil certain roles or types (multi-typing is possible). The simplest version of this concept is depicted in Figure 4.3. This setup is more advanced and flexible than the simple, direct typing. Using the concept of inheritance (sub-typing) even more complex mappings are possible.

Dynamic/Relative Mapping

No matter how well structured a complex mapping even with roles and subtyping may be, reality is always more complex and especially dynamic. In an organization/execution system there are more complex relationships/associations than the sub-type/"is-a" relationship, that may also be of relevant for process execution. E.g. a business process for a vacation/holiday request with one Subject called *"Employee"* the other *"Supervising Manager"* as shown in Figure 4.4.

In an organisation with multiple hierarchical tiers, an agent *"Marry"* could be the supervisor for *"Paul"* and be able to authorize his vacations, while having to request her vacations in principle form *"Mark"*, who is her supervisor. Ideally, whenever some agent/subject-carrier initiates the vacation request process as an employee, the Subject *"Supervising Manager"* should automatically be mapped to her or his specific current supervisor. Furthermore, if that specific agent is not available but there are rules to determine a deputy or substitute, the subject

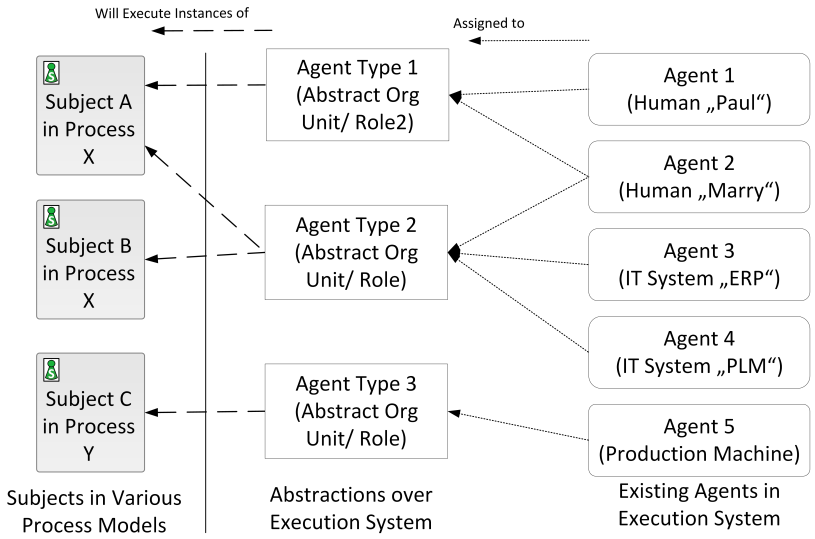


Figure 4.3: Complex Static Mapping of Subjects and Execution System Agents via Groups/Roles (without sub-typing)

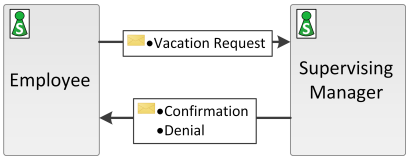


Figure 4.4: SID for a Vacation Request Process

should automatically be mapped to that substitute. E.g. if *"Mary"* is currently on vacation herself and *"Paul"* needs to request a vacation, e.g. *"Mark"* could be there to fill in the role of *"Supervising Manager"* to *"Paul"* while she is away.

In these cases, the actual mapping of subject to agent is **dynamic** and depends on the the current status of the execution system. For it to work without human interaction, two things are necessary that need to work jointly:

- First there must be a way to model the execution system or organization formally and define various relations, rules, and associations between possible agents/users. Furthermore the current status of the execution system must be kept track of in this kind of data base. (e.g. currently available agents).

- Secondly, there must be a formal way to formulate *relative mappings*. E.g. *"Subject B is executed by an agent that has the role of supervisor to the agent that executes Subject A"*.

- Lastly, a system is needed that can be queried with the relative mappings and evaluate them in **context** of the current status of the execution system. Such a query is answered with a set of available agents that fulfill the queried term.

Therefore, such a dynamic mapping is only working in a more complex setup or process system that is depicted in Figure 4.5.

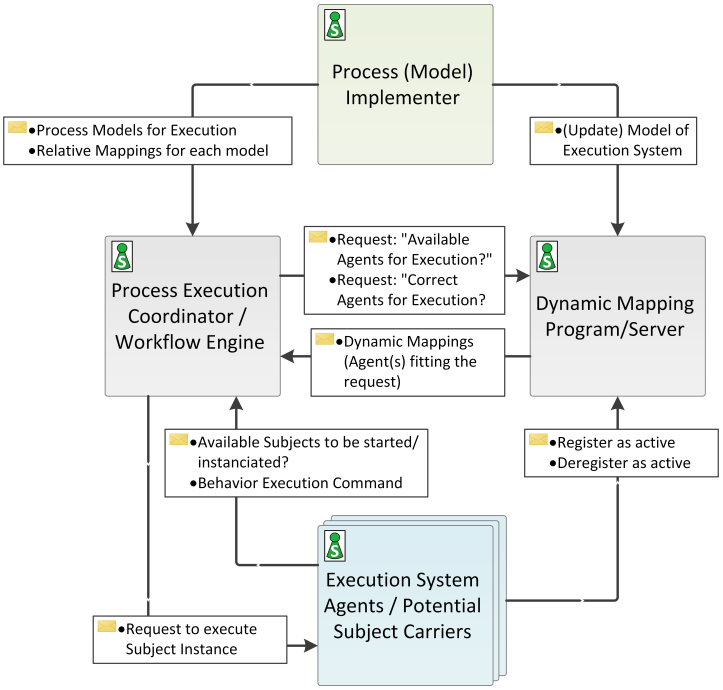


Figure 4.5: SID for a Process (Model) Execution system with Dynamic Subject-Agent Mapping

While relative mappings in a dynamic scenario are in principle more abstract than the model of execution system, they still need to be written in accordance with it. E.g. concepts or types like "supervisor" need to exist in the execution system model in order to be used in expression for relative mappings. Consequently, whenever the implementation of processes needs to be distributed between different execution systems there need to be means to have at least partially a standardized description model and vocabulary.

4.1.2 Implementing Objects

The previous section was concerned with mapping the subjects of a process model to the agents of an execution system. Next to Subjects, a PASS process model also contains definitions for (data) objects in the form of messages and subjects' data stores. As far as the model is concerned, PASS does not distinguish between **physical** objects without an organizational impact (e.g. work pieces), purely **logical** (data) objects to be exchanged within IT Systems, and objects that are physical and logical nature at the same time (e.g. a check list on paper). Messages can be used to represent all three. Within the means of PASS, they are distiguished by the type of **Payload Description** the Message Specification has. It will either be **Payload Data Object Definition** , usually in an IT interpretable form, or a **Payload Physical Object Description** that most likely has an human readable form to it an describes what the physical object represented by the message is made up of.

If no explicit and formal differentiation is done in the model, then it is also part of the implementation to implement the messages correctly. E.g., it must be determine whether a message *"Quality Check List"* is indeed printed out and worked on as a piece of paper, or if and how this check list is indeed realized as an IT-Artifact as part of a digital work flow system.

Naturally, there is always a close relationship between subject and the associated objects. E.g. messages from a subject "Deburring Machine" that receives "unfinished work pieces" and sends "finished work pieces" are very likely to exist as actual physical objects, while a "Status Report" from an "Application Server" subject is very likely to be digital data sent or displayed on a computer.

4.2 Modeling an Organization

Especially in a formal/software context, in order to map the subjects in a process model to "agents" of an execution system, it is necessary to describe or model the according socio-technical execution system itself.

This section describes the principle features, concept and concerns, necessary for a modeling approach that can be used to formally describe the organizational view of a company / execution system / organization — A modeling approach different form PASS, an approach meant to describe an organization/system and not the processes running in it! The language in principle should be to represent any kind of resources of a company/organization (people, software, machines etc.) and their arbitrary relationships amongst them. This way, the organizational structure of a company can be mapped very precisely.

Technically, these kind of descriptions can be realized with many means. Due to their Ontological nature, RDF/OWL could be used and SPARQL, a query language for linked data, could be employed for the expressions. Nevertheless, here only the principles are defined.

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].

Let's start with the essential question whats makes up an organization.

4.2.1 What is an organization?

Insights:. We are looking at 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 4.6.

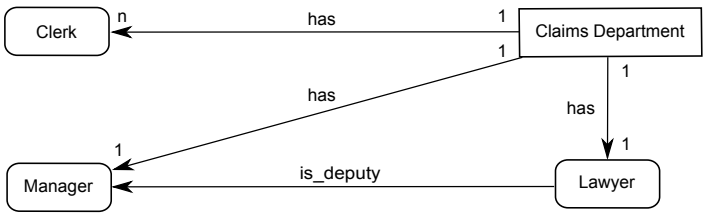


Figure 4.6: Claims department in general

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 4.7). 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[2], the

[2]We are using fantasy names.

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 4.7).

Looking at this two departments we also found an interesting mutual deputyship between the lawyers of the two departments (cf. figure 4.7). 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).

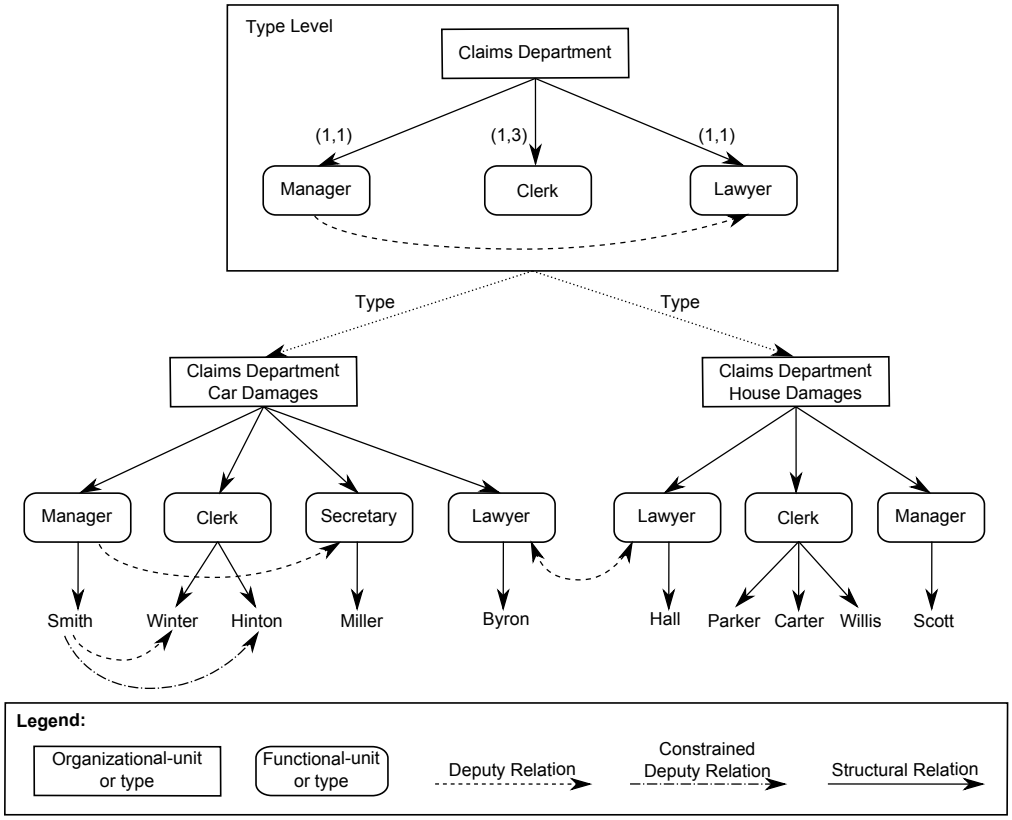


Figure 4.7: Type and instance level of the example, adapted from [LSR13, fig. 3]

Lessons Learned. This section describes additional observations concerning real world organization policies[3].

Knowledge Hierarchy. As we have seen there are different levels of organizational knowledge. There is a level of general structural assertions like *"a department consists of one to three clerks"*. 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* [of the principle idea of departments] — things become more concrete and specialized.

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

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.

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. Consequently the Organisational model becomes a complex knowledge graph — much more than simply hierarchical tree-structure.

4.3 Formal Specification of Organizations and Dynamic Mapping Algorithm

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 ([Sch98]) .

The model is comprised of structural definitions for models that can be used to describe an organization, as well as definitions of algorithms, that are supposed to be executed upon that structure in order to allow for a generic mapping.

For the structural specifications standard Mathematical Set-Notation is used. For the algorithms a generic mathematical-pseudo-code notation is being used.

Within the 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:

4.3.1 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"’.

4.3.2 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 $name \in \mathcal{BEZ}$.

4.3.3 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 $\Gamma_s^{\Upsilon'} = \Gamma_s^{\Upsilon} \rhd \mathcal{OE}$[4]. 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[5]. 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:

[4]The operator $\rhd$ is defined as $((\Gamma \subset A \times B) \rhd (C \subset B)) := \{(x, y) \in \Gamma \mid y \in C\}$.
[5]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.

4.3.4 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.

4.3.5 Dynamic Mapping Algorithm

As discussed in the sections for dynamic mapping (4.1.1) such a dynamic mapping server ideally has 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.

4.4 Physical infrastructure

As mention actors can be humans as well as machines or software components. We will therefore look at how the concept presented in the previous chapters works in an area of with almost no human actors. Let us take a look at the Industry 4.0 area.

Production plants as we know them today are actually overthought within the smart factories initiative. Central production plans for manufacturing big numbers of similar products are replaced by intelligent objects embedded in self-organizing systems called smart factories [Gro15]. These objects are cyber-physical systems [Mei13] using intelligent sensors for gathering information about the world around them. They are able to react to environmental changes and to generate plans for fulfilling their goals. So, if a customer orders a product at a manufacturing company, an agent responsible for the production of the article is created. This agent knows all about the bill of material and the working plan for the creation of the product. On basis of this information the agent generates a plan how the product can be manufactured. For this he has to talk other agents representing the resources of the company.

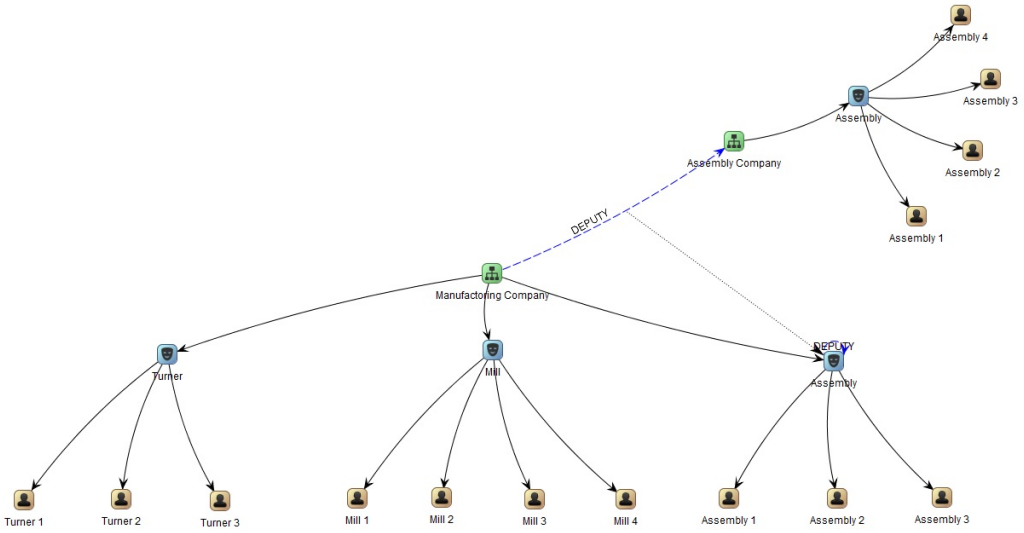


Figure 4.8: Joint Organizational Model that incorporates two Partner Companies

Figure 4.8 depicts an excerpt of the organizational model of a manufacturing company producing wooden chairs. The model encompasses the internal organizational unit *Manufacturing Company* with the subordinate internal functional units *Turner*, *Mill* and *Assembly*. The internal resources *Turner 1* to *Turner 3* are related to *Turner*, *Mill 1* to *Mill 4* belong to *Mill* and *Assembly 1* to *Assembly 4* are part of the internal functional unit *Assembly*.

The properties and capabilities of the elements of the organization can be described using attributes. In our example, the maximum length of a workpiece to be processed and the shape of a workpiece on the machines is of interest. Table 4.1 describes an excerpt of attributes in conjunction with their values.

Looking at the relationships, there exists an external partner company (assembly company) that can take over assembly tasks on load peaks [6]. The fed-

[6]The automated propagation of model elements (entities, relations and attributes) to partner

Resource	Attribute	Value
Turner 1	minworkpieceLength	30
	maxworkpieceLength	50
	workpieceKind	octagonal
	...	...
Turner 2	minworkpieceLength	35
	maxworkpieceLength	40
	workpieceKind	octagonal
	...	...
Turner 3	minworkpieceLength	40
	maxworkpieceLength	50
	workpieceKind	square
	...	...
Mill 1	maxLength	100cm
	maxWidth	100cm
	maxHeight	20 cm
Mill 2	....	...
Assembly1	Type	304456
	...	...
Assembly2(PC)	Type	304456
	...	...

Table 4.1: Attributes of Resources

eration between the manufacturing company and the assembly company only works if there is a combined/joint organizational model. The external organizational unit *Assembly Company* encompasses the external functional unit *Assembly* in conjunction with the external resources *Assembly 1* to *Assembly 4*.

Dynamically finding the correct or at least fitting agent for a specific task is basically done the same way as before by resolving an dynamic mapping expression compatible with combined execution system model. The expression can reference entities, relations and/or attributes. Suppose a production agent wants to have 64 chair legs manufactured for the production of a batch of chairs of type 304456. He first looks in the master data record to see what specifications the legs must have. He finds a length of 40 cm and octagonal shape. He can use these two specifications to find a suitable machine. The expression looks like this:

```
(Turner)(Manufacturing Company). ATT.(minworkpieceLength ≤
"40" AND maxworkpieceLength ≥ AND workpieceKind = "octagonal")
```

This means the production agent is looking for an agent of type "'turner"' that is able to produce workpieces with length 40cm and an octogonal form.The search is local to the manufacturing company. In our example, turning machines 1 and 2 are eligible for the production step. Finding a production resource to manufacture the 64 seats and backrests works on the same principle. Now the chairs still have to be assembled. The agent finds a suitable assembly unit with

organizations is described in [LRS14].

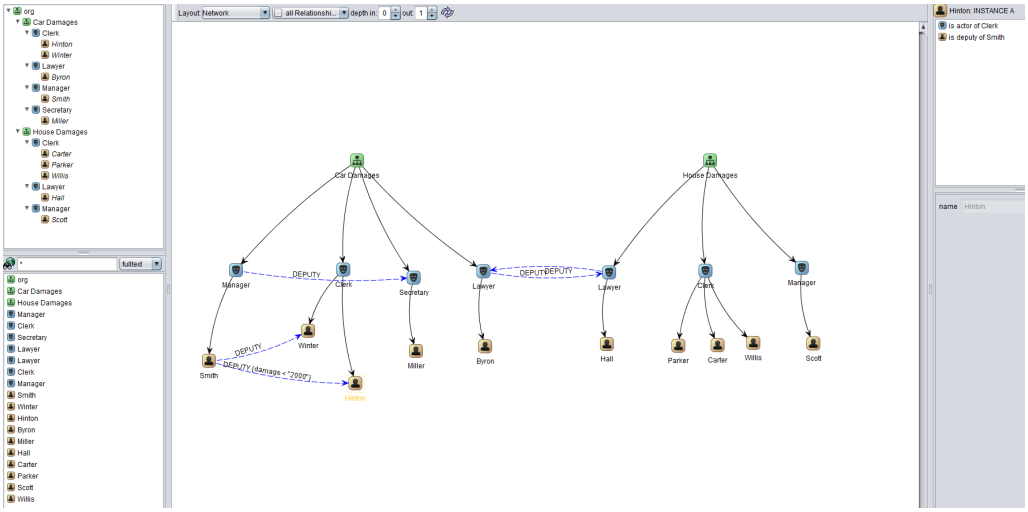


Figure 4.9: Screenshot: Implementation of the $\mathcal{C} - \mathcal{ORG}$ GUI

the following query:

```
    (Assembly)(*).ATT.
(Model = "304456"))
```

The asterisk means that substitutions are also permitted for this request. Thus, the company's own production facility (Assembly 1) as well as the machine of the partner company (Assembly 2) can be found. Depending on the degree of workload of the resources, the agent decides whether he then locks the order on his own machine or the machine of the partner company.

Similar to production resources, software products, actuators or sensors can be modelled and managed according to the same principle.

4.5 Practical Realization

The formalism discussed in this contribution is implemented in a prototype. Figure 4.9 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 section and 4.3.5. 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 4.10 provides an enlarged view of the model editor[7]. 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[8] 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.

[7]It shows the example model (cf. fig. 4.7). The type level is hidden.
[8]The relation-types to be shown can be selected.

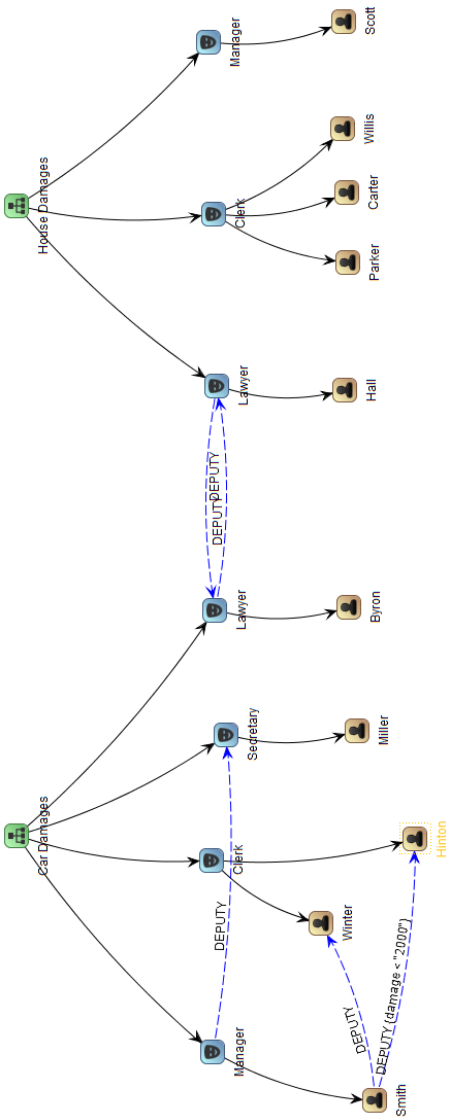


Figure 4.10: Model Region of the Implementation

CHAPTER 5

Various Aspects for further Standardisation Activities

In this chapter is a loose collection of various aspects of relevance for the paradigm of subject-oriented for process modelling and programming. These aspects have been published as different conferences contributions. The following sections are based on these publications. The concepts described in theses sections will be part of future standardisation activities. For now, the following sections are not aligned and should be considered as the independent works that they are.

CS: Hochkommas und - im pdf falsch gedruckt.

CS:They contain original text parts and thus, the conclusions need to be aligned to the standardization effort, as tried for Fog Computing.

The following sections are based on following publications:

- Subjects and Shared Input Pools: [FS19]
  Subject-Phase Model based process specifications [Fle13]
- Hierarchies in Communication Oriented Business Process Models: [EF19]
- Business Activity Monitoring for S-BPM: [SF13]
- Subject Oriented Project Management: [AWC14]
- Subject-oriented Fog Computing: [SFS18]
- Activity based Costing [ZSF13]

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 The subject "Car detection" represents the shared input pool.

CS: hier fehlt das originäre Beispiel als Bezugspunkt.

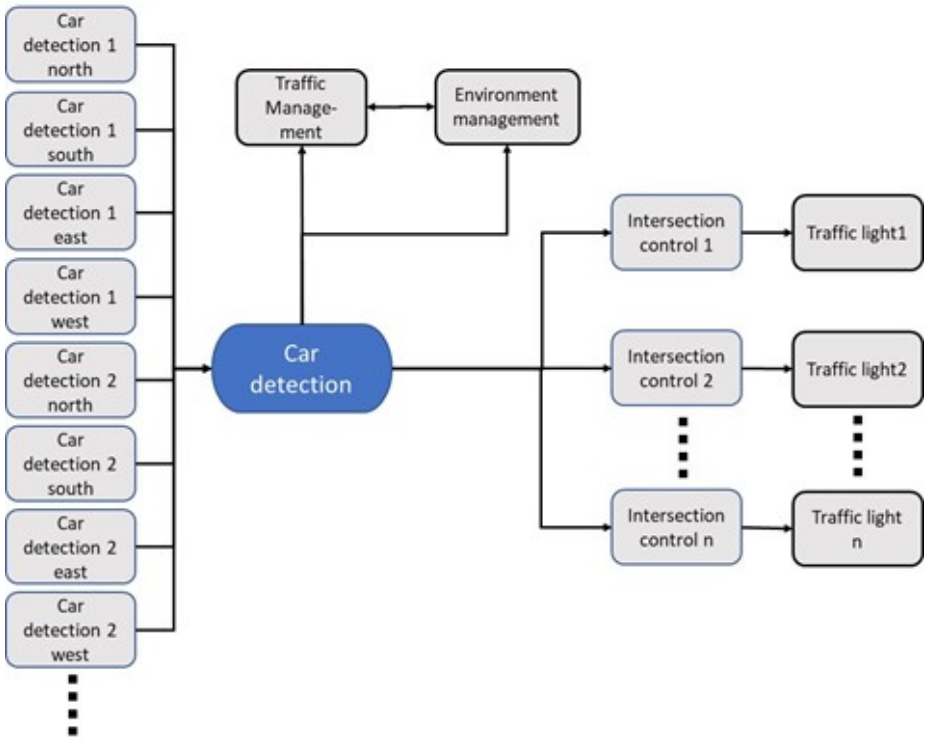


Figure 5.1: Traffic Management System with Shared Input Pool

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.

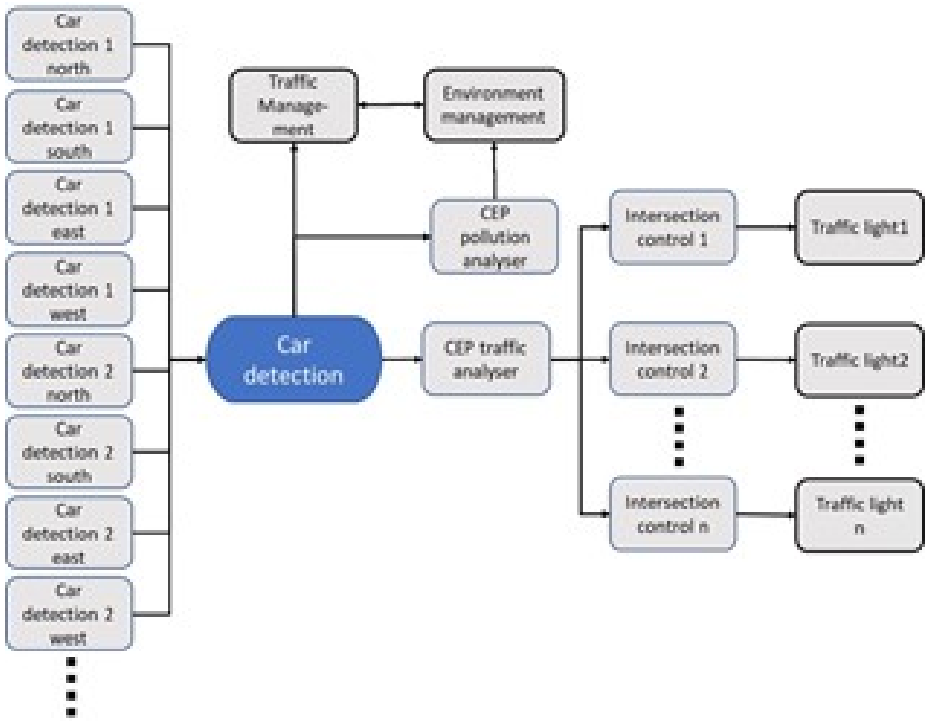


Figure 5.2: Shared Input Pools and Complex Event Processing

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 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 [Dre17].
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 [Wal19] 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 [ECF+18].
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 [Fou19] 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 [DCL+17] 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 [CGK19].
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 [XWS19]. For flexibility reasons we prefer off chain oracles - see figure 5.3.

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

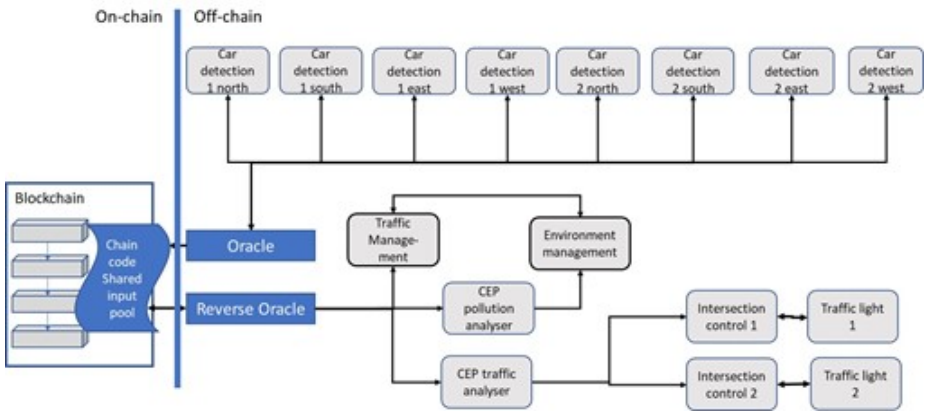


Figure 5.3: Utilizing block chain patterns Oracle and Reverse Oracle

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 Subject-Phase Model based process specifications

5.2.1 Introduction

There are different stakeholders in business process management (see e.g. in [FS11], [ASM09], [vBM10]). The most important ones are the customers interested in the result of a process, managers responsible for processes (often called process owners) and the parties involved in the execution of a process (Providers). The parties involved in the execution of a process (providers) can be represented by subjects (see [et.11]. Subjects are abstract representations of the active elements in a process which communicate with each other in order to coordinate their work producing the required result of a process. Especially middle management wants to get an overview about the process from the event causing the execution of a corresponding process instance (e.g. subject representing the customer of a process) till the results of a process are delivered to the parties expecting it. In the most abstract way, upper management is only interested in Key Performance Indicators expressing the efficiency and effectiveness of a process. May be that the delivery of a process result causes the execution of a succeeding process. These succeeding processes use results of the proceeding process as input. Middle management is interested in who is involved in a process and what is their contribution to the process result and what are the cost for each contribution. Managers are not mainly interested in the details how the parties involved in a process execute their tasks, how they communicate with each other in order to coordinate their work and which tools and means they use to do their work. Whereas the parties involved in the execution of a process are exactly interested in these aspects of a process. The parties involved in the execution of a process want to know what work they have to do, in which sequences they have to execute these tasks, including when they have to communicate with whom about what and last but not least which means they use for doing their tasks. On the one hand processes have to be described precisely enough that the parties involved in a process execution know what they have to do in which situation and on the other hand management is primarily interested in more abstract process attributes like key performance indicators and the structure of a process, depending on the management level.

Additionally abstract views on processes are needed if complex process systems have to be defined and a top down approach for describing processes is applied. First designers create an overview of a process system and then this abstract model is refined step by step till a model is created which is good enough for the parties working in a process. In order to define hierarchically most methods and tools for process management use different approaches to solve that problem. In that article an approach is described which allows to get an overview of the involved parties of a process and what are their major contributions to the result of a process. Practical experience show that this approach allows to describe the dynamic os a process on one page and the specification is precisely enough to derive executable workflows.

5.2.2 Related Work

Many approaches exist in order to express hierarchies of processes in business process management. In general three to five levels of process specification are used (e.g. see page 53 in [ASM09], page 52 in [HP05], page 92 in [SN11]) These

process levels are mainly called process areas, business processes, processes, sub-processes and activities. Sometimes sometimes business processes are classified in kernel processes, and main processes (see [SN11] page 87. In the following sections the abstraction concepts in the mostly used process modeling approaches are outlined.

In the ARIS (see [Sch01], [ea01], [SN11]) ecosystem process chains are used in order to give an overview of a process. A process chain shows the process of a process system and defines the sequence in which these processes in the process system can be executed. There can be several levels of process chains. This means an element representing a process of a process chain can also contain a process chain and so on. In the lowest level each element of a process chain contains a Event driven Process Chain (EPC). The event driven process chain contains the activities executed in a process. On EPC level the activities are connected to subjects who have to execute an activity. This means only at the lowest level a manager can see who is involved in the execution of a process. It is possible that an activity in an EPC is not one single activity. It can again be an EPC. These various levels of abstraction are focused on the activities executed in a process.

BPMN has pools and swim lanes for describing process structures (see [OMG18]). A process consists of one or several pools and each pool can consist of one or several swim lanes. It is not allowed to have pools in a pool or swim lanes in a swim lane. Activities are assigned to swim lanes. This means there are three abstraction layers: Pools, Swim Lanes and Activities. Pools and swim lanes are used to structure the activities in a process. An activity can contain a sequence of activities. This activity type is called subprocess. An activity in a subprocess can also be a subprocess and so on. This means on activity level arbitrary levels of subprocess are possible. In subprocesses pools and swim lanes are not allowed. Each activity is assigned to a party executing that activity. This means that a pool or swim lane does not correspond to a certain doer. Each activity in a swim lane can be executed by a different provider.

This two ways of abstraction are similar to the abstraction used in control flow charts. Only at the lowest level management can see who is involved in a process. The other way around the parties executing the actions have to scan through all the activity sequences in order to find the actions which they have to execute.
In this paper I want to show an approach in order to specify processes on different abstraction levels for the major stakeholders: Management and doer or provider. On one page managers and the actors in a process see the structure of a process (people and activities) and they get an overview who has to execute which action when.

5.2.3 Subject Phase Matrix (S-PM)

The features of the subject phase matrix will be demonstrated with an example from a real process management project in a small transport company. Figure 5.4 shows part of the process architecture of that company. There are three processes which built a process chain. There is the process 'Order'. In that process the order of customer will be checked and the transport will be prepared. In process

'Transport' the goods will be transported and in process 'invoicing' the message 'invoice' is send to the subject 'customer' and its payment is controlled.

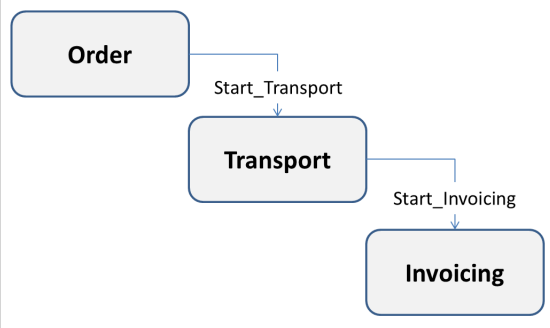


Figure 5.4: Process-architecture

The execution logic of each of these processes is described with a subject-phase matrics (S-PM). The following figure 5.5 shows the S-PM of the process order.

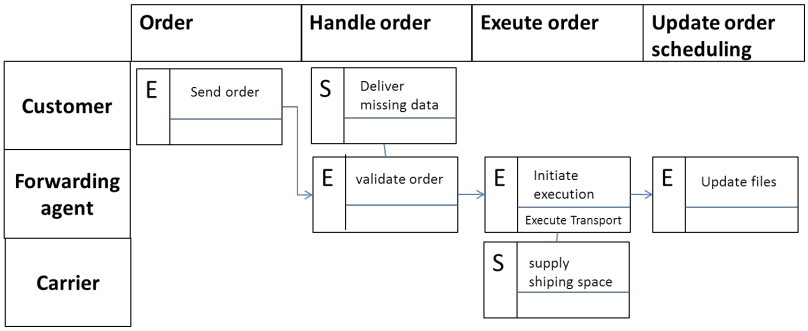


Figure 5.5: S-PM of process 'Order'

A S-PM shows the subjects of a process in the most left column. Subjects represent the acting parties in a process. Subjects are abstract resources like in S-BPM (see [et.11]). The subjects in Figure 5.5 are 'customer', 'forwarding agent' and 'carrier'. An S-PM does not show the embedding of a process into an organization. This is a completely separate step as described in [et.11]. This means in that article only the process model is considered. The implementation activities are analog to S-BPM (as described in [et.11]). Process 'order' shown in 5.5 has the phases

- order
- handle order
- execute order
- update scheduling

A process is executed from the left to the right, but loops back to proceeding phases are allowed. This means a process starts with the most left phase. In the columns representing the various phases of a process the activities executed in that phase are specified. In a phase it is defined which subject executes which major activity set (marked by an E) and which subjects execute supporting activities (Marked by an S) for the executing subject. In figure 5.5 the subject 'Customer' executes the activity 'send order' in phase 'order'. In phase 'Handle order' the subject and 'Forwarding agent' are involved. The subject 'Forwarding agent' validates the order and the subject 'Customer' supports in that action. In order to get this support the subject 'Forwarding agent' communicates with the subject 'Customer'. In phase 'Execute order' the subject 'Forwarding agent' is supported by the carrier. In that phase subject 'Forwarding agent' starts the process 'Execute Transport' (Lower part of the rectangle representing the major activity in phase Execute order). If phase 'Execute Order' is finished subject 'Fprwarding Agent' executes in phase 'Update Order Schedulin' the activity 'Update Files'. Figure 5.6 shows the S-PM of process 'Execute Transport' initiated in process 'Order' in phase 'Execute Order' by subject 'Forwarding Agent'.

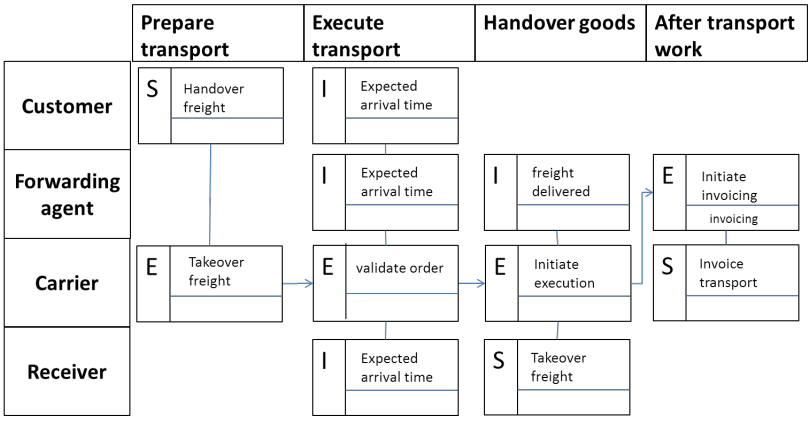


Figure 5.6: S-PM of Process 'Execute Transport'

In process 'Execute Transport' some activities are marked with an I. This means that the corresponding subjects are only informed. In phase 'Execute process' subject 'Carrier' informs the subjects 'Customer', 'Forwarding agent' and 'Receiver' about the 'Expected arrival time'.
In the S-PM 'Execute Transport' there is also a subject named 'Customer'. This subject is different from the subject 'Customer' in the S-PM 'Order'. Subject names in a S-PM are valid in the corresponding process. This means subjects in different processes with the same name are different subjects. This does not mean that during the execution of a process (process instance) subjects with the same name in different processes are handled by different persons. Which providers (persons or machines) executes the activities of a subject are defined in the organisational embedding. This is not considered in that paper. This is a task of embedding subjects in an organization (see [et.11]).
S-PMs give an overview about the subjects involved in a process, which activities they execute in which sequence and which relations exists between the various subjects. In many real projects (ISO 9001 projects) more than hundred processes have been specified in that way. It showed up that each process can be structured in phases and processes have between three and six phases only a

small number (around 2 percent) have more than six phases. In all cases S-PMs did have more than one page. There has been also the experience that S-PMs are easy to understand also for management and gives the involved parties a first impression what they have to do in a process. In spite of their overview character S-PMs are precise enough that a S-BPM specification can be automatically derived. This will be shown in the following section.

5.2.4 Conversion of Subject Phase Matrix to Subject Communication Models

The conversion of the Subject-Phase-Matrix into Subject Communication Diagrams (SCD) and Subject Behavior Diagrams (SBC) can be done automatically (see [et.11]). In general these SCDs and CBCs are executable without any additionally programming (see [et.11]) but the business objects must be added manually and some internal activities must be specified in more detail. In the following the focus is on the behavior of subjects. Data or business objects are not considered.

In order to transform an S-PM into a subject-communication specification we consider the matrix from the perspective of the involved subjects (see figure 5.6). Each subject in the matrix corresponds to a subject in the subject communication

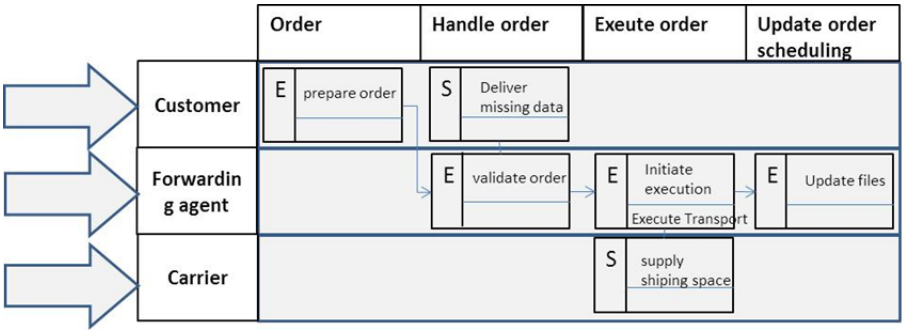


Figure 5.7: S-PM considered from the subject view

diagram. The names of the subjects in the S-PM are extended with an Id for the process. In our case to each subject name the letters AO are added (AO=Accept Order). A subject sends a message to another subject if in the S-PM the E-action in the succeeding phase is executed by a different subject or the subject needs supports from an other subject. If subjects send a message to the subject executing the E-activity in the succeeding phase the message is named 'E-name-of the-succeeding-phase'. If a subject requests support from another subject the message is named 'S-Name-of-the-phase?'. The receiving subject sends the result of the support request back with a message called 'S-Name-of-the-phase!'. Figure 5.8 shows the communication structure derived from the S-PM shown in figure 5.5.
Subject 'Customer-AO' is the start subject. It is the only subject which executes activities in the start phase of the S-PM Order. The succeeding phase 'Handle order' is executed by subject 'Forwarding Agent' therefore subject 'Customer' sends the message 'E-handle-Order' to subject 'Forwarding agent-AO'. This subject executes the activities in phase 'Handle Order' (see figure 5.5).
Figure 5.9 shows the behavior of subject 'Customer-AO'. In order to see the re-

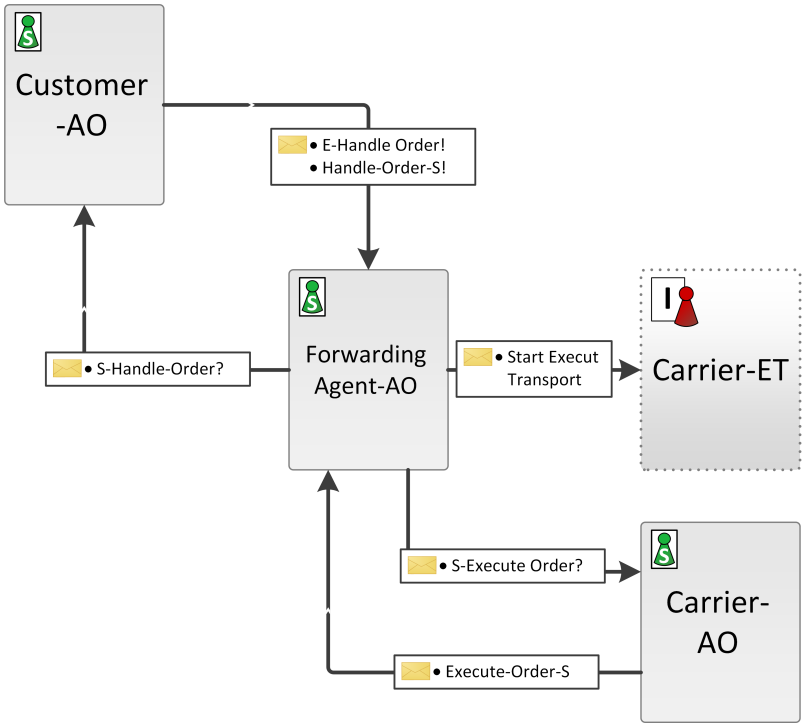


Figure 5.8: Subject Communication Diagram of S-PM of Process 'Order'

lationship between the phases in the S-PM and the activities in the SBD the activities in the SBD belonging to certain phase in the S-PM have frames marked with the phase name. After the activity 'prepare order' is finished the next phase

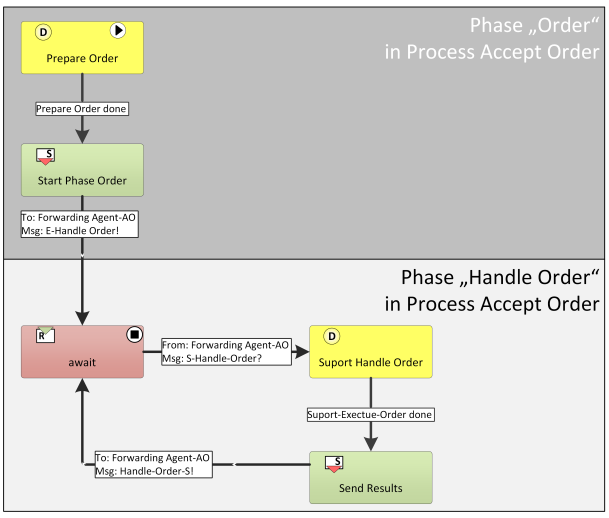


Figure 5.9: SBD of Subject 'Customer-AO'

is started by sending the message 'E-Handle-Order' to the subject 'Forwarding Agent-AO'. Than subject 'Customer-AO' is waiting for the message 'S-Handle-Order?' from subject 'ForwardingAgent-AO'. This is a support request which is sent by subject 'ForwardingAgent-AO' in phase 'Handle order'. The support activity is executed and the result is sent back to subject 'ForwardingAgent-

AO'. The following figure 5.10 shows the more complicate behavior of subject 'ForwardingAgent'. The subject 'ForwardingAgent-AO' receives the message 'E-Handle-Order' from the subject 'Customer-AO' in phase "Handle Order" in the S-PM. After that message the subject 'ForwardingAgent-AO' checks whether it need some support from the subject 'Customer-AO'. If support is required a corresponding support request message ('S-Handle-Order?') is sent to the subject 'Customer-AO'. After an answer has been received the subject 'ForwardingAgent-AO' continues its work and checks whether some additional support is required or the activities in that phase can be finished. If that phase is finished the subject 'ForwardingAgent-AO' continues its work in phase 'Execute Order'. Because the subject 'ForwardingAgent-AO' is also the executing subject for phase 'Execute order' it is not necessary to send an E.message to the executing subject of the succeeding phase. In phase 'execute-orde' the subject 'ForwardingAgent-AO' starts also the process 'Execute-Transport' by sending the message 'E-Start-Execute-Process' to the subject 'Carrier-ET' which is a subject in process 'Execute Transport'.

Figure 5.11 shows the behavior of subject 'Carrier-AO'. In process 'Accept Order' the subject 'Carrier-AO' only supports subject 'ForwardinAgent-AO' in phase 'Execute Order'. This means the subject 'Carrier-AO' receives a support request message 'S-Execute-Order?', excutes the support activities and send the result back to the subject 'ForwardingAgent-AO' by the message 'S-Execute Order!'.

Figure 5.12 shows the SBD of process 'Execute Process' which is derived from the corresponding S-PM (see 5.6). The mechanism for getting the communication structure is the same as shown with process 'Accept Order'. The external subject 'ForwardingAgent-AO' represents the starting subject in process 'Accept-Order'.

Figure 5.13 shows the behavior of subject 'Carrier-ET' for the first two phases of the corresponding S-PM shown in Figure 5.13. This subject is different from the subject 'Carrier-AO' but it can be handled by the same person. But this is a decision during the embedding of a process into an organizational structure (see [et.11]).

5.2.5 Evaluation of the Transformation

S-PMs combine SCD and SBD. It show the subjects as active elements in a process, which subjects communicate with each other and which are the major phases of a process.

The phases can be seen as subprocesses producing some interim results. The author do not know a general proof that all processes can be structured in phases, but many widely used process specification methods use process phases (see section 'related work'). In his practical work the author hasn't yet found a process which could not structured in 3 to 6 phases, sometimes up to 8 phases (around two out of hundred).
The conversion of each process phase into SCD and SBD is based on modeling by restriction (see chapter 6 in [et.11]). Modeling by restriction is executed in 5 steps:

- Identify the number of subjects involved in a process,
- give these subjects names which describe their task in the process,

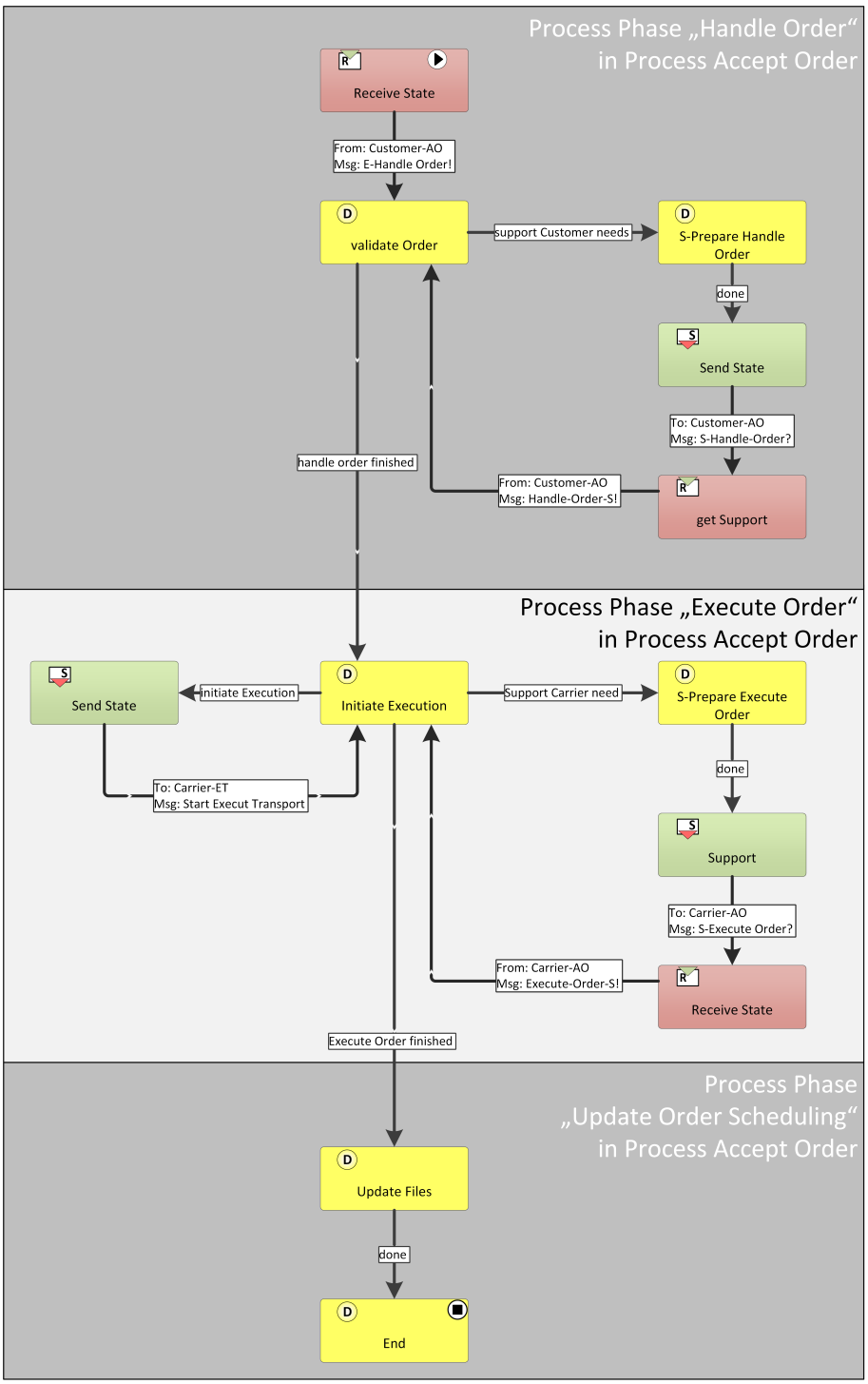


Figure 5.10: SBD of subject 'Forwarding-Agent' in Process 'Accept Order'

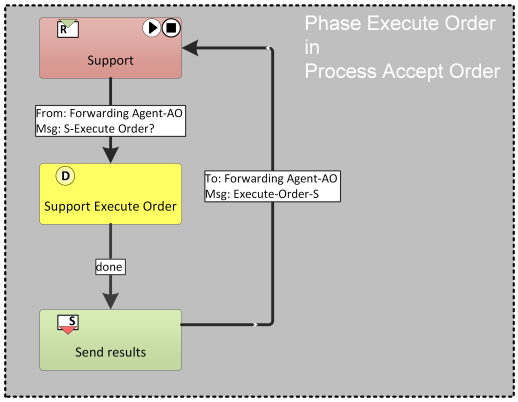


Figure 5.11: SBD of subject 'Carrier-AO' in Process 'Accept Order'

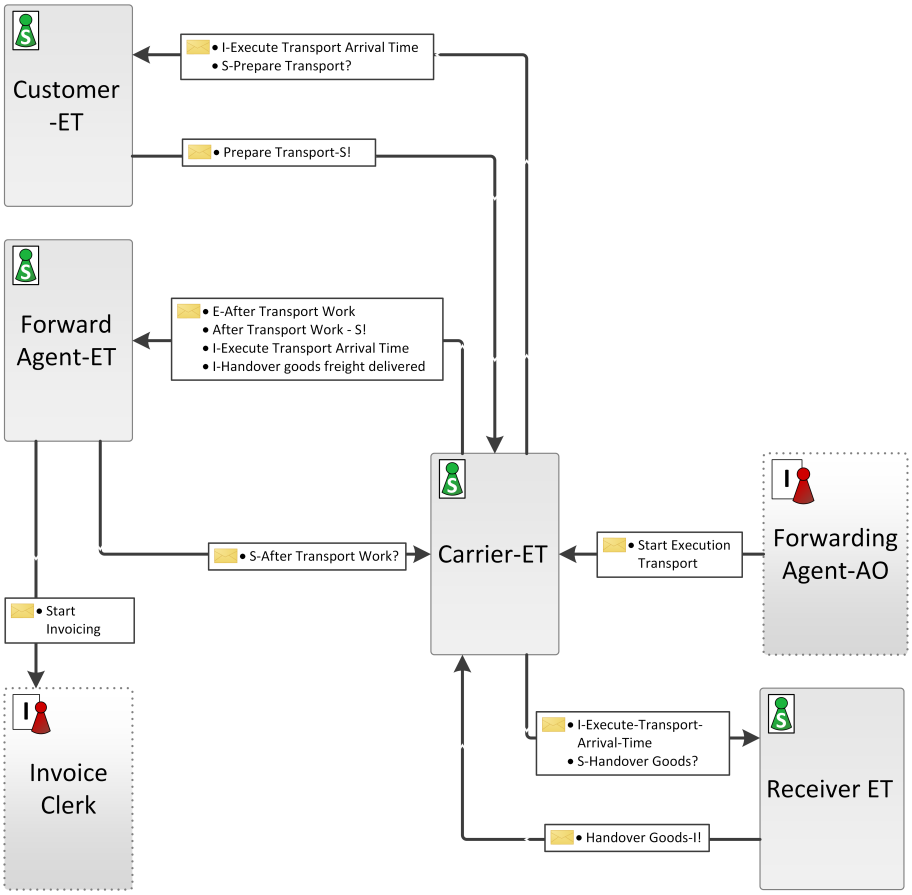


Figure 5.12: SID of Process 'Execute Transport'

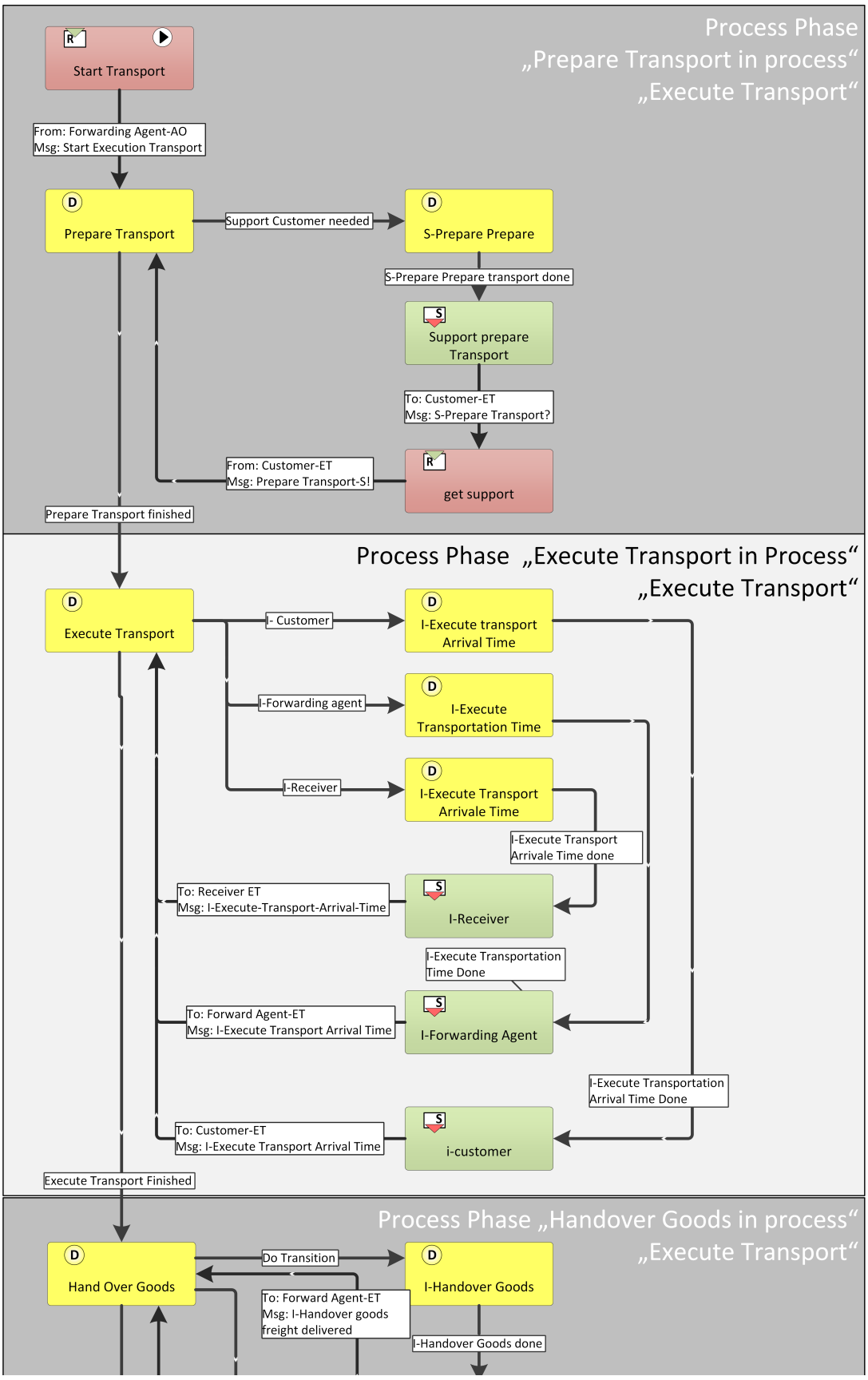


Figure 5.13: SBD of Subject 'Carrier-ET' in Process 'Execute Transport'

- remove not allowed communication paths between subjects,
- introduce process specific message names
- adapt communication behaviour to process requirements

An S-PM contains the information to execute the first 4 steps automatically for each process phase.In a phase the E-Subject communicates with the S- and I-subjects in a not very strict way. This means an E-Subject can request a service from a S-subject as often it wants, which may be not correct in the sense of the considered process. Finally an E-Subject decides whether a succeeding phase is started. This means a S-PM is converted in a process consisting of a chain of not very strictly defined subprocesses derived by modeling by restriction from the S-PM.

5.2.6 Activity and Data Details in S-PM

In S-PM mainly the sequencees of the execution of the major activities are considered. In order to describe data and the details of the activities executed in a process phase a so called Input Execution Output table (IEO-Table) is used. The following figure 5.14 shows the IEO-table for the S-PM of the process 'Accept Order'.

	Order	Handle order	Execute order	Update scheduling
Input	Required transport capacity	Order	Complete order	Transport in execution
Activity	Prepare order	Key in order; Agree on terms and conditions;	Procure transport capacity; Initiate transport execution	Update transport schedule
Output	order	Complete order	Transport in execution	Updated transport schedule

Figure 5.14: Input Execution Output Matrix of Process 'Accept Order'

The columns represent the phases of a process as in the corresponding S-PM. In the input row the data required in a process phase are specified, in the activity row the activities executed in a phase are listed and the output row contains the results of a process phase. The activities listed in the activity row define the details of the internal activity called 'phase name' or 'S-phase name' in a SBD derived from a S-PM. Based on that information the SBD can be described in more detail. The input and output row can be used to define the business objects used by the various subjects and transported by the different messages. In order to develop a systematic way to hand over that information to more detailed SCDs and SBD some additional work has to be done and experience has to be gathered. Up to now only a very intuitive approach is used.

5.2.7 Conclusion

S-PMs give managers and subjects involved in process an overview of a process provided that a process can be separated in phases. Practical experiences over

more than 10 years show that this is possible without any difficulties. S-PMs are precisely defined which allows to convert them in SCDs and SBDs automatically. This conversion is based on modeling by restriction. This automatic conversion result in a first version of SCDs and SBDs which must be more restricted by the requirements of the considered process and enriched with the required business objects. Details for a S-PM are described in so called IEO-tables which contain details about the required data, the executed activities and the results of a process phase. In further work it has to be investigated how this infomation can be used in automatic conversions.

5.2.8 Future Work

Several aspects and topics need to be addressed by future research: spacing

- Definition of structural semantics in OWL
- Formal definition of the transformation from a Subject-Phase Matrix to a subject oriented model

5.3 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.

5.3.1 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 5.15 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 5.15: High level structure of car break down service

Figure 5.16 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".

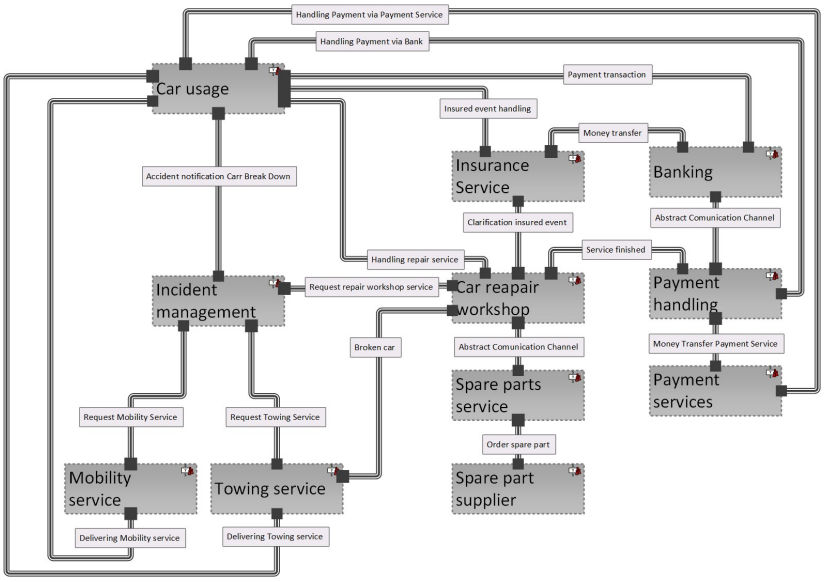


Figure 5.16: Structure of the Emmergency Call Handling Process

Inside a process there can be also processes. This means that levels of processes can be built. Figure 5.17 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".

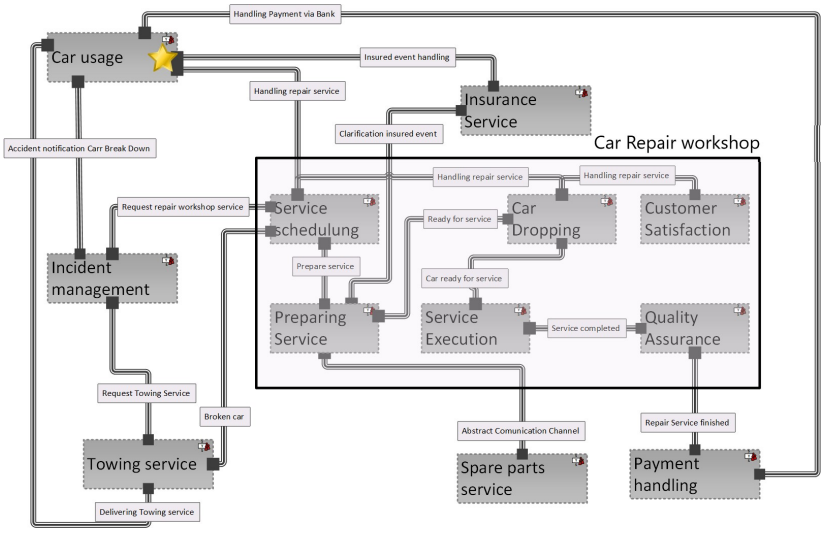


Figure 5.17: Details of the "Car repair workshop" Process

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 architec-

ture. 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 5.18 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".

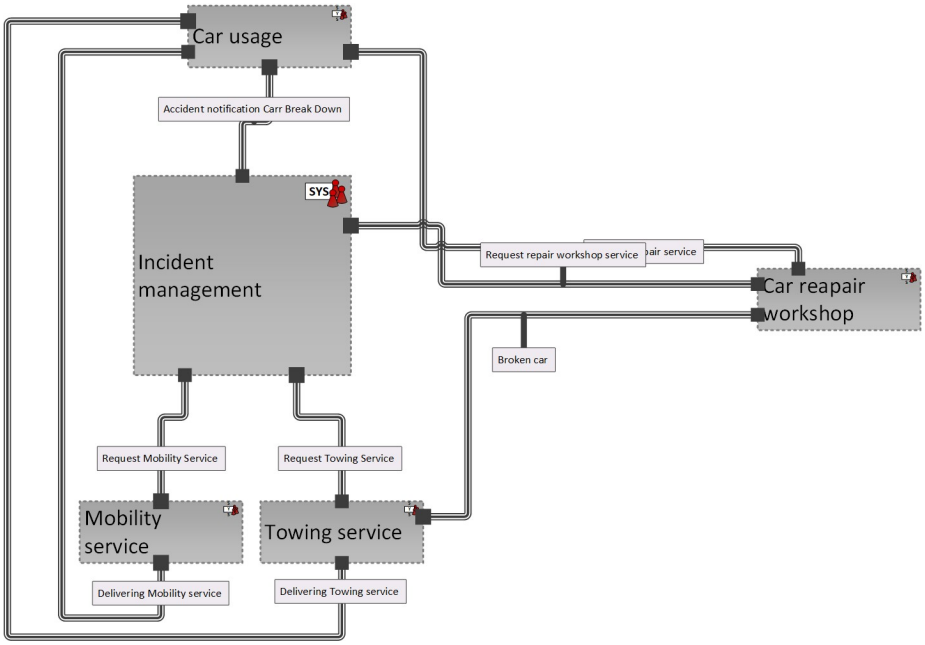


Figure 5.18: Neighbors of the "Incident Manaement Process"

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". The process "Incident management" with all the border subjects is shown in figure 5.19.

The border subjects of the processes "Mobility service", "Towing service" and "Car repair work-shop" have the same name “Service agent” 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"

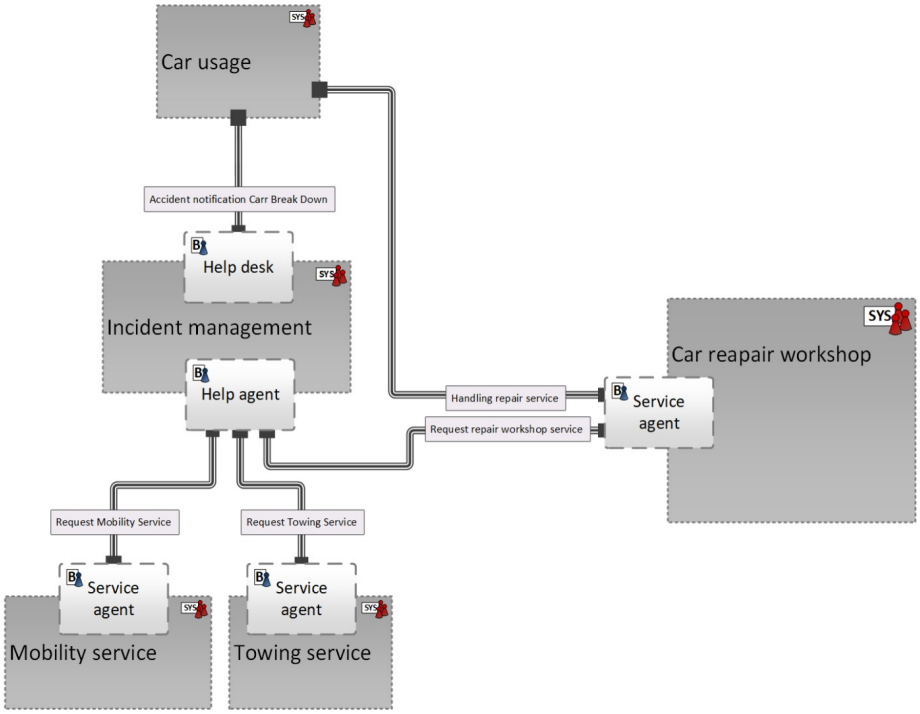


Figure 5.19: Border subjects of the "Incident Management" Process

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 5.20shows the Subject Interaction Diagram of the process incident management.

5.3.2 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 5.21 shows the border subject "Help desk" of the processes "Incident Management" which communicates with the border subject of process "Car usage".

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 [FSS+12b]) of the process “Incident Management”. Figure 5.21 shows the detailed Subject Interaction Diagram around the subject help desk.

Instead of the channels the messages required for a towing service request are shown. A message "Request towing service" comes from the interface sub-

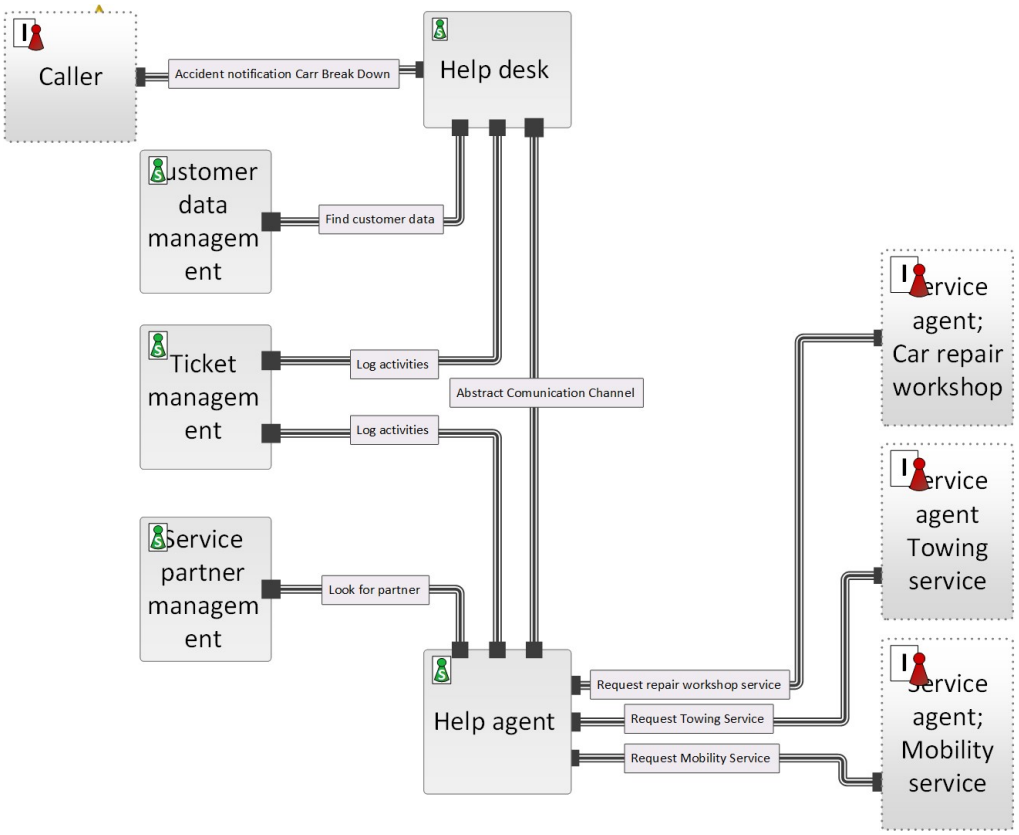


Figure 5.20: Subject Interaction Diagram of the Process "Incident Management"

ject. 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 5.22. 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 5.21.

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

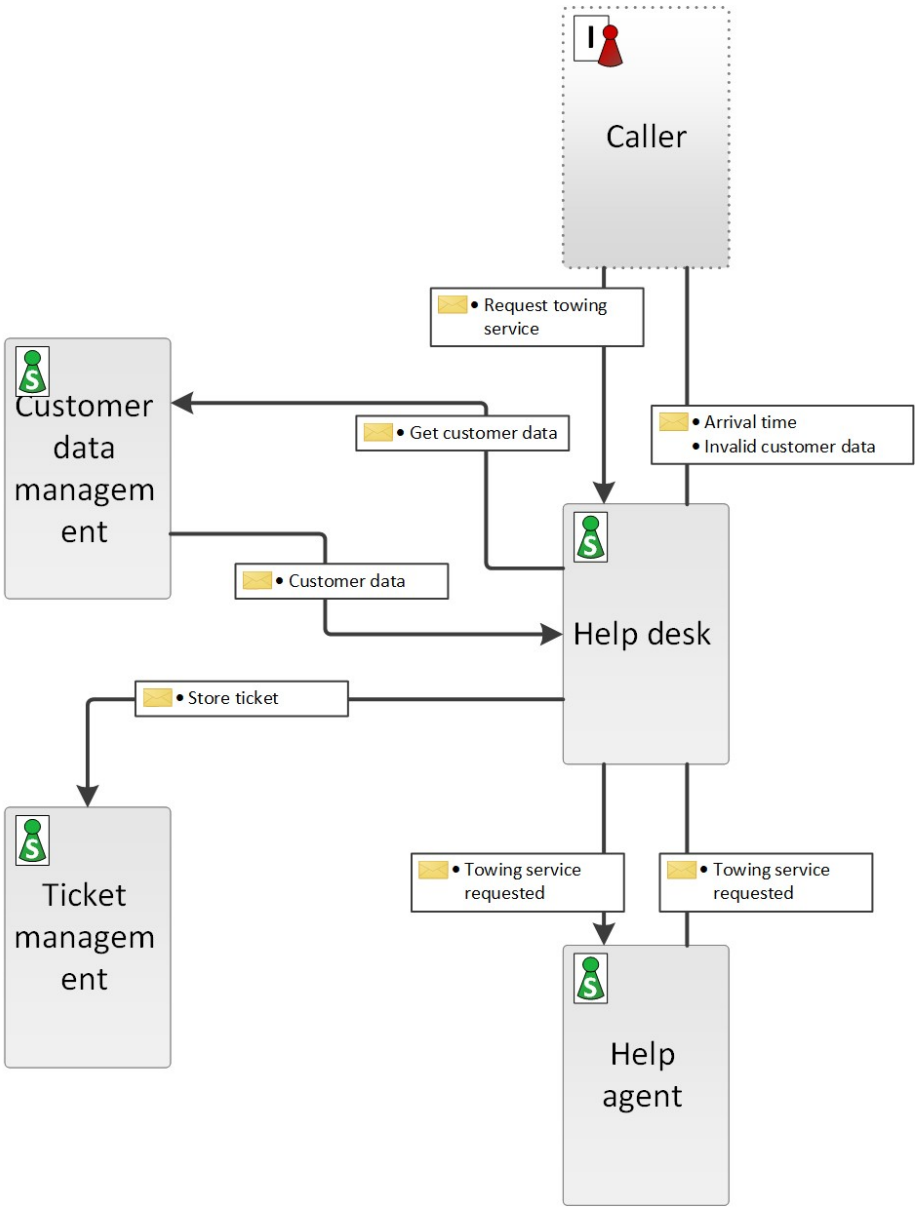
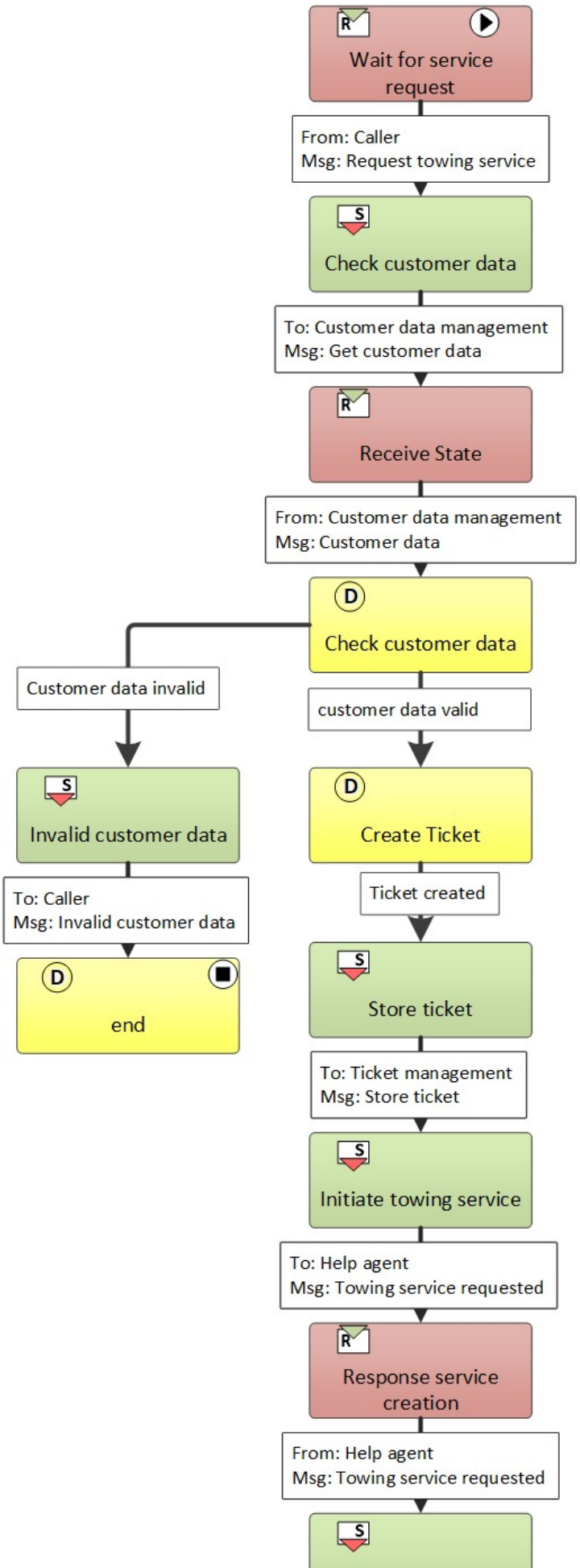


Figure 5.21: Subject Interaction around the subject “Help desk”


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

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 5.23 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".

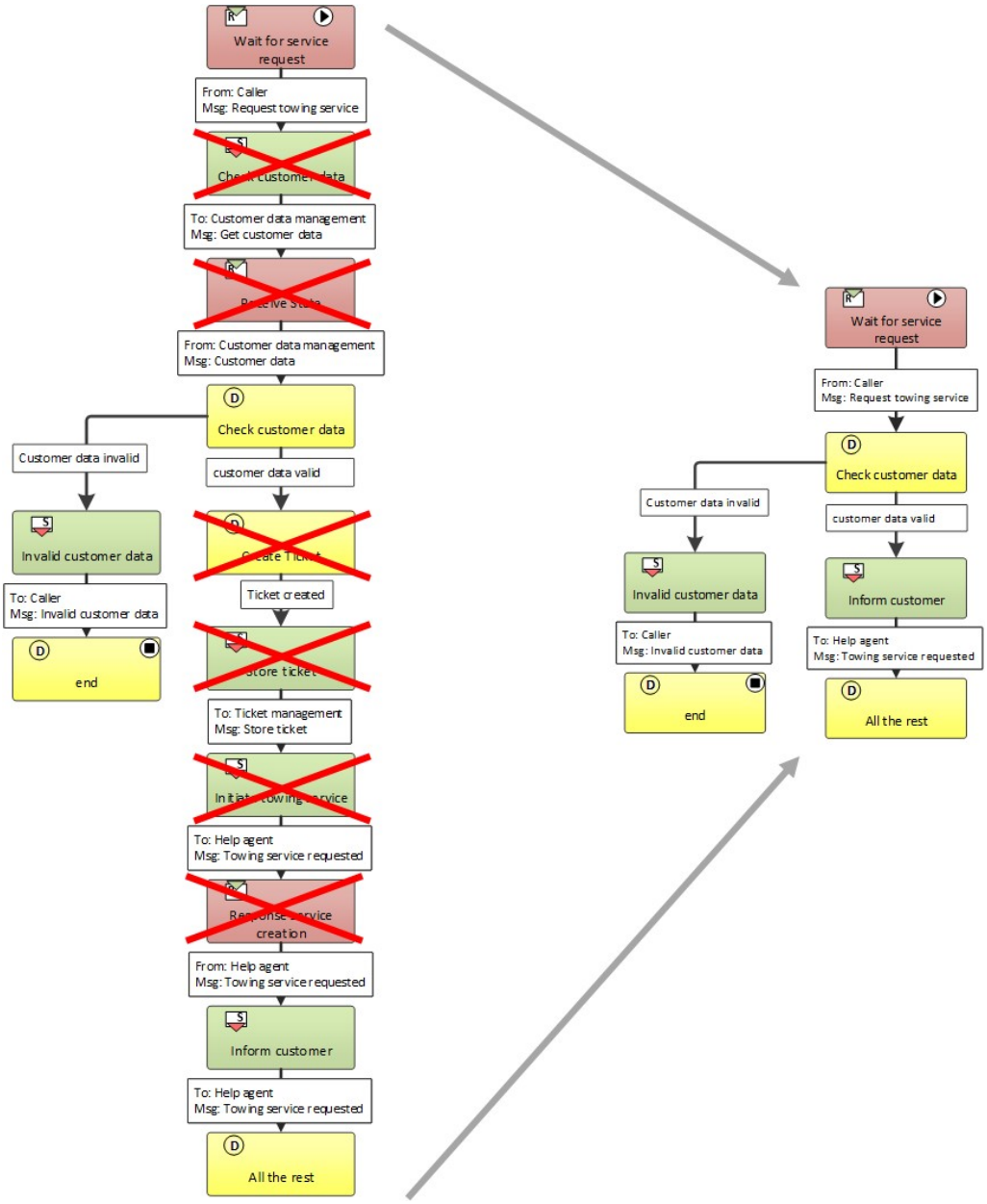


Figure 5.23: Deriving the Behavioral Interface from the Subject Behavior

A behavoral interface is always relative to a communication partner. In figure 5.23 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 [MFR+10]).

5.3.3 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.

5.4 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 (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 [Hes05]. 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 [SAO09] and thus helps implementing the operational level in the closed-loop approach to enterprise performance management [Sch13] (see figure 5.24).

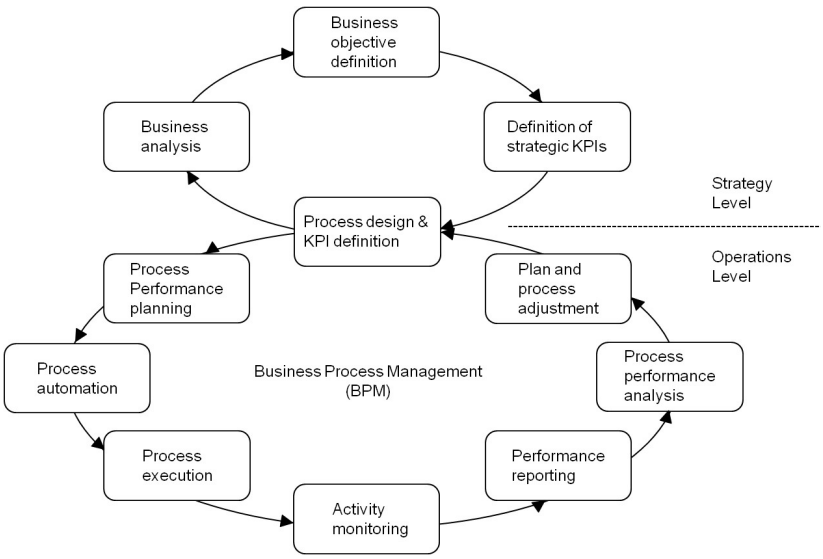


Figure 5.24: Closed-loop Approach to Performance Management [Din06]

5.4.1 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 5.25) and [Sch13], [EN11] , [JMM11a]). 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 dash-

boards 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.

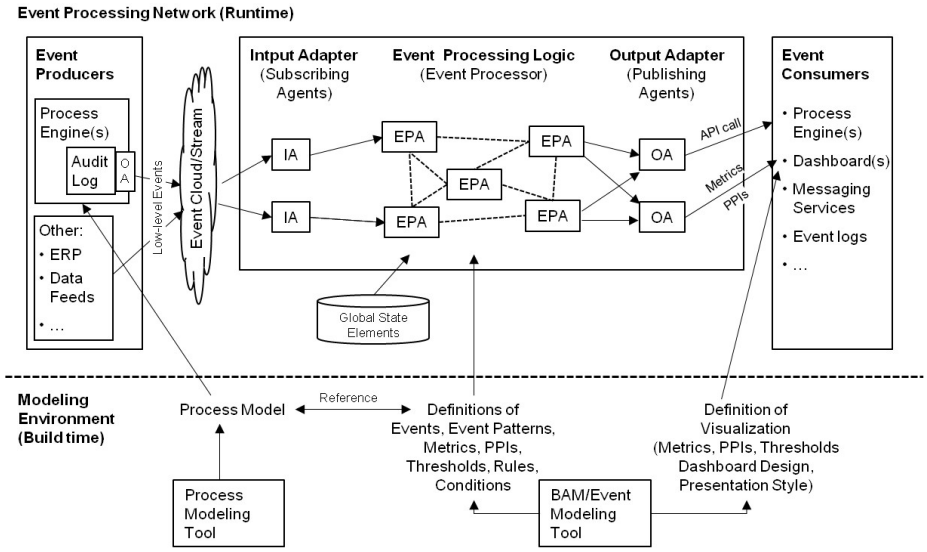


Figure 5.25: Integrated BAM/CEP Architecture [Sch13]

5.4.2 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 [SS10]. 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 [Kue09], [GJH09]. Figure 5.1 shows an extract of a sample fact sheet defined for the average processing time of activities (see also [ZSF13], [Peh12]).

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 [DRODRDTRC12] 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 [FJMM12]. 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) [FJMM12]. 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 5.26 de-

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 defined as the average of Duration of Analyse RFC activity
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 be slower than or equal to 1 working day
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>	Event logs of BPMS
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 [DRODRDTRC12]

picts 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 5.27. 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 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

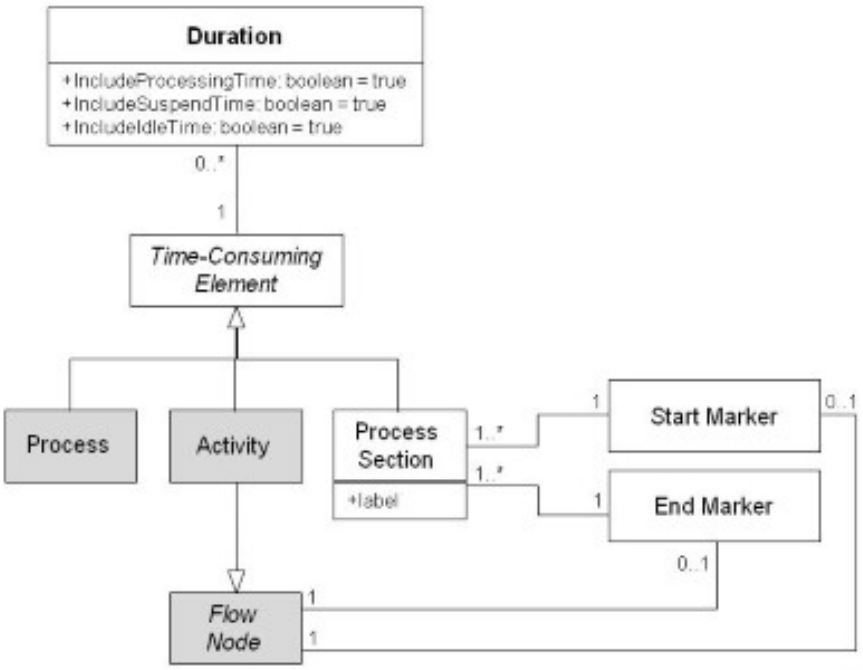


Figure 5.26: Meta Model for Duration (related to BPMN) [FJMM12]

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 [Boe12] the mapping should be possible without problems. Table 5.3 compares the BPMN specification elements used by [FJMM12] with the ones appropriate in S-BPM [FSS+12b].

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 ob-

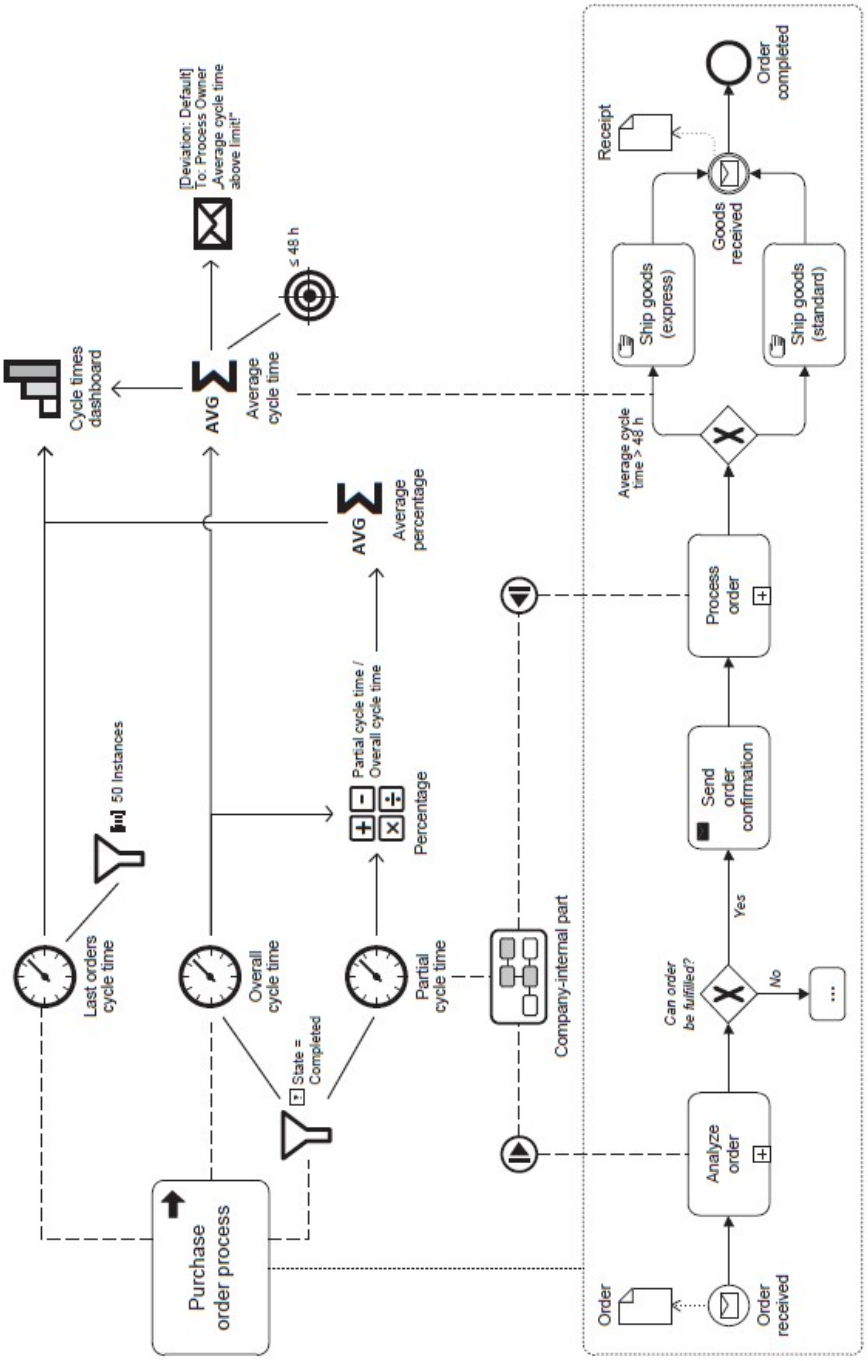


Figure 5.27: BAM Model for Cycle Times of a Purchase Order Process based on BPMN [FJMM12]

jects as equally important parts of a process, offers the subject as an additional specification element to add . In figure 5.28 we modified the picture of figure 5.26 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.

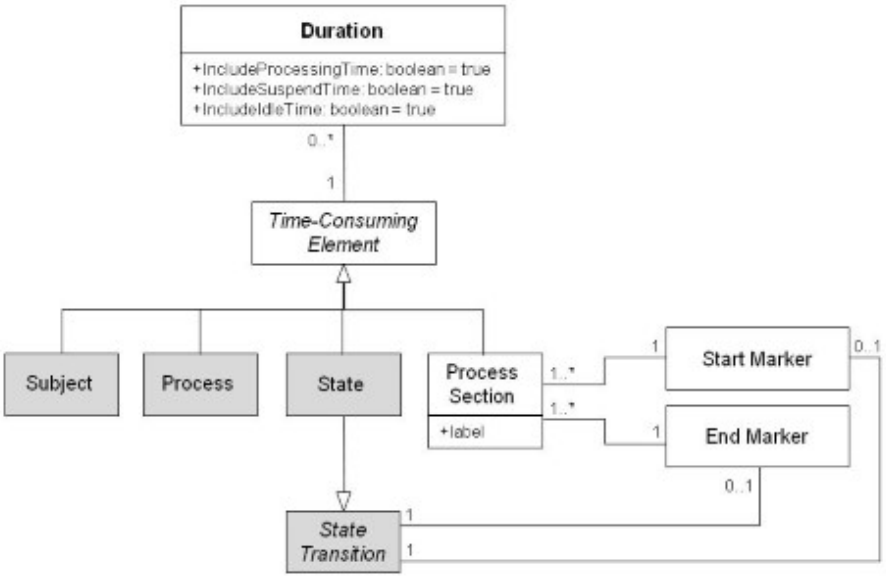


Figure 5.28: Meta Model for Duration (related to S-BPM)

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 5.28. 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 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 5.30 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.

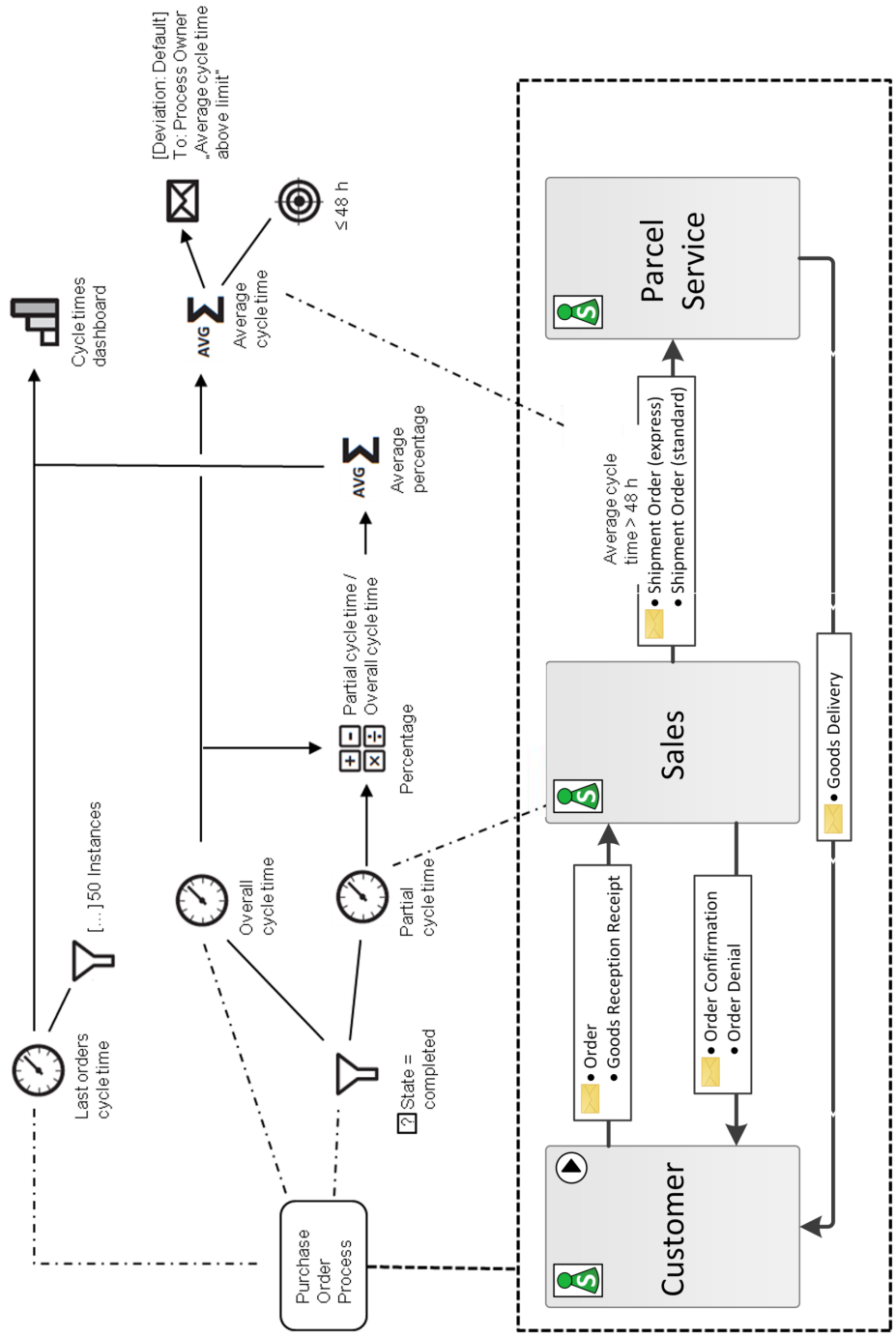


Figure 5.29: BAM Model for Cycle Times of a Purchase Order Process based on S-BPM

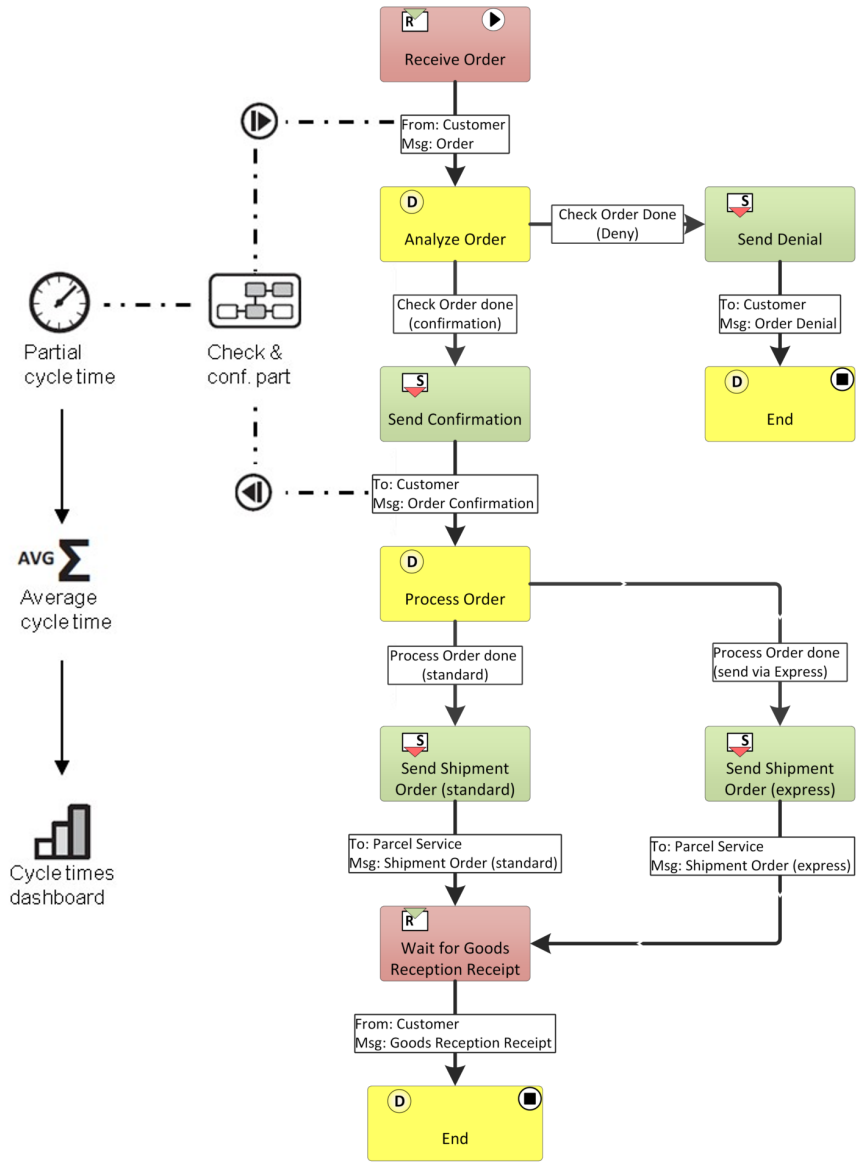


Figure 5.30: BAM Model for Cycle Time of a Process Section based on S-BPM

5.4.3 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.

5.4.4 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
- ASM definition of execution semantics for throwing events if process performance indicator boundaries are violated.
- ASM definition of execution semantics for handling violation events

5.5 Subject-Oriented Project Management

Subject orientation is focused on networks of independent 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 [JS98]. 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 5.31 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.

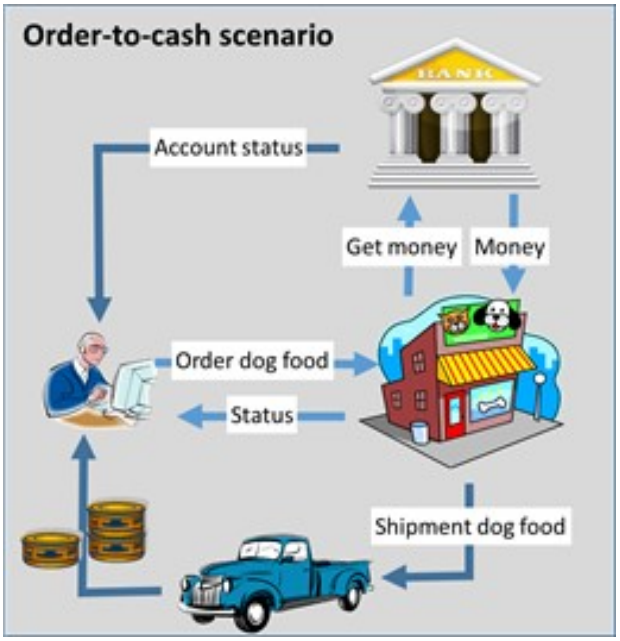


Figure 5.31: Order-to-cash scenario in a federation of enterprises and applications (simplified)

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. [Gru02]). 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 pro-

posal of a software development approach for federated systems based on subject orientation.

5.5.1 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. [HFP03] , [LLX09]):

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

[JS98] 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" [DEG03]. It may include modeling concepts, tools for model-driven architecture, integrated development environments (IDEs) etc. The so-called magic triangle (see figure 5.32) summarizes the various aspects of a software development methodology [JH13].

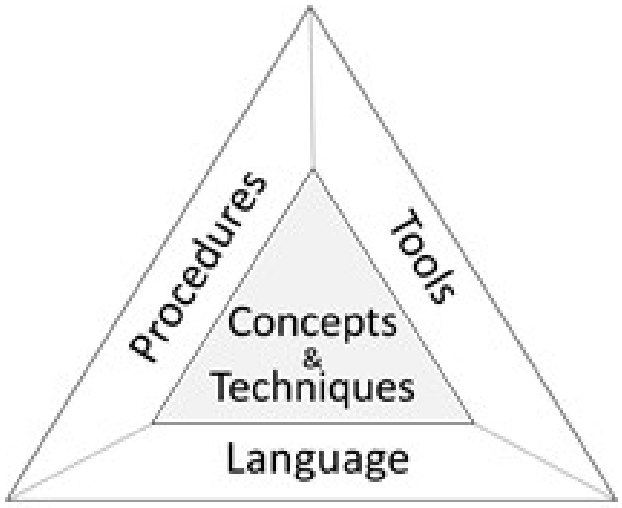


Figure 5.32: Magic triangle of software development methodologies

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 environment.

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 soft-

ware 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. [RRF08], [JH13] and [DEG03] 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. Conway's law states "organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations" [Con68]. A variation of Conway's 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 architecture" [JONB04]. 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.

5.5.2 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 [JONB04]. 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 5.33 (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.

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

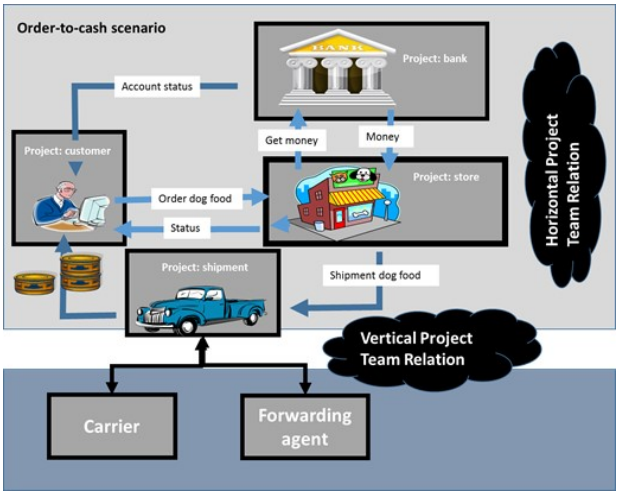


Figure 5.33: Multiple-team project and its communication structure

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 "Transfer order" 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 5.34.

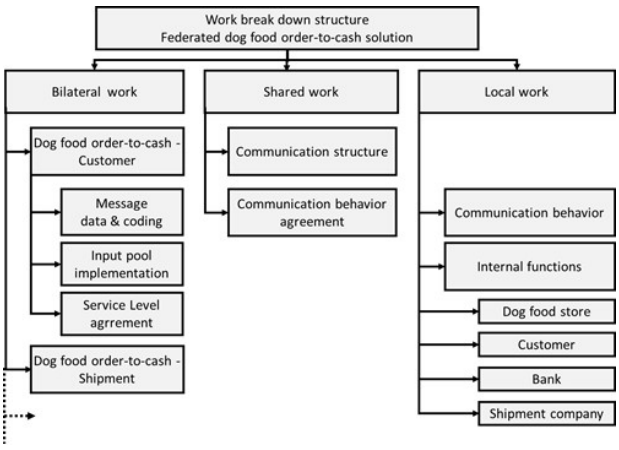


Figure 5.34: Work break down structure for the development of a federated system

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.

5.5.3 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.

5.5.4 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.

5.6 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 [YL15]. 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 [SDW+15],[HXW13].
Such demands pose significant challenges to existing development paradigms, particularly in terms of edge computing and stakeholder-oriented communication capacities (cf. [YL15],[SDW+15]]). 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 [SDW+15]. Besides network technologies, mobile cloud computing is a typical enabler for this demand [HXW13].
However, according to Syed et al. [SFI13] 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 [BMNZ14]: 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.

5.6.1 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 [SFI13]. 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 [Jam11]. 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. [AGT+17]).

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. [FSS+12b]), 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. [AGT+17]).
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.

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

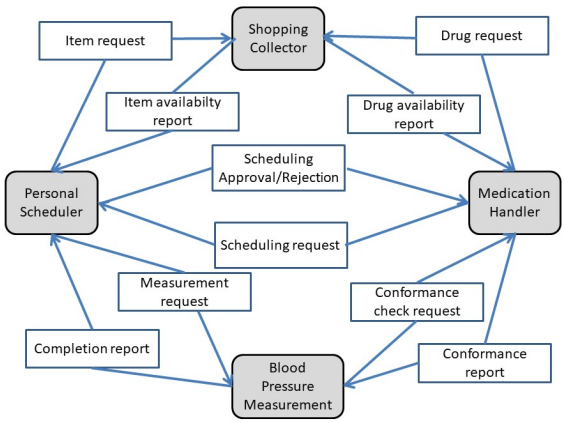


Figure 5.35: Example of home care support (SID)

(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. [FSS+12b]). 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 [KSW16] 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.

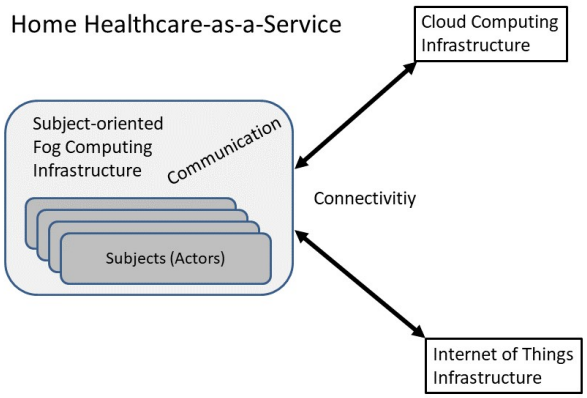


Figure 5.36: Fog Computing Architecture

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, 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 'local' 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. [BMNZ14]), 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. [VRM14a], 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. Ac-

cordingly, 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 5.35 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.

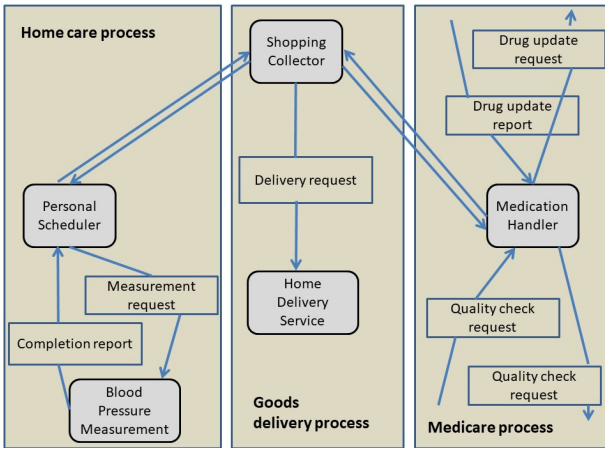


Figure 5.37: Extended subject interaction diagram for the process 'home care'

For SFC implementation the open source engine UeberFlow [KS16] 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.

5.6.2 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. [VRM14b]), leading to more situation-sensitive Fog applications (cf. [GDG16]).

5.7 Activity Based Costing

CS: table refs are incorrect

5.7.1 Basic Concepts

Process Controlling

Process controlling has both a strategic and an operational dimension [cf. e.g. [SS10], 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 5.24 in 5.4). 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. [Sch13], [JMM11b]).
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. [SS10] 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 5.38 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. [SS10] 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).

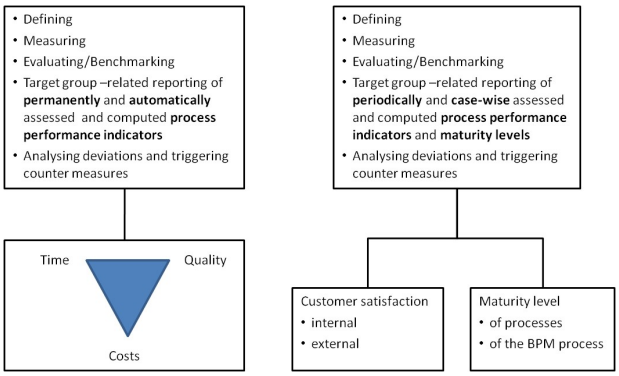


Figure 5.38: Operational Process Controlling

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 [MV85] and Cooper and Kaplan [CK88] and was established in the German-speaking community by Horvath und Mayer [HM89].
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 5.39)

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

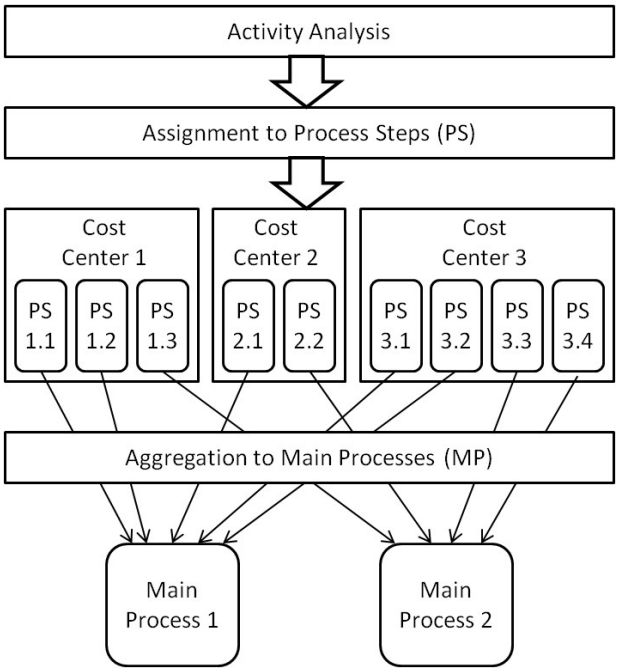


Figure 5.39: Process Structure

- 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 [BS97] 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 5.7.2.

5.7.2 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 [FSS+12b], [SFG09].
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 5.7.2).

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 5.40 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 [SF13]. For clarity reasons in the figure 5.40 we only added time stamps for one state and just added the duration for the others.

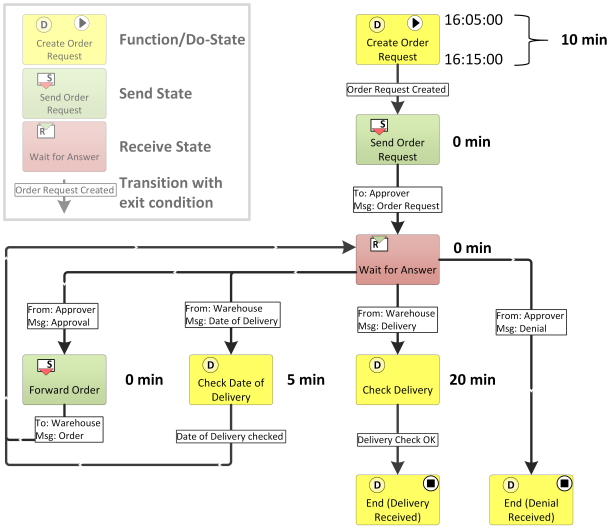


Figure 5.40: Calculating Processing Time of the Subject 'Purchaser'

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 5.40). 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 5.5 gives an overview of the rates for the employees involved as subject representatives.

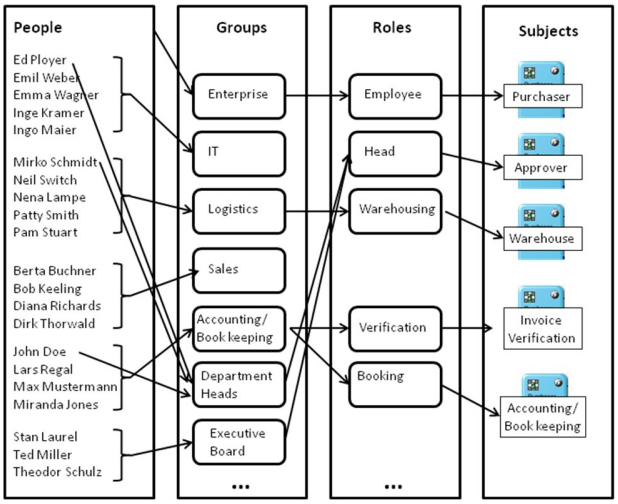


Figure 5.41: Embedding the Subjects into the Organization

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 5.6). Considering a sufficient number of instances over a representative period of time allows computing a valid average cost value.

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

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

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

Table 5.6: Personal Related Process Costs

the processes and process steps it is involved in. Table 5.7 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 [Kue09]. Table 5.8 depicts such a fact sheet created for our purposes [Peh12]. A more formal structure can be found in [DRODRDTRC12].

5.7.3 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.

5.7.4 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.

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	Sum up Processing time * hourly wage rate Sum up 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'

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.

5.8 The Arbitrator Pattern for Multi-Behavior Execution

This section describe the principle for a PASS execution that can cope with multi-behavior subjects. It is an updated version of [ESF12]. The proposed interpreter is formulated using the *Abstract State Machine* (ASM) concept.

5.8.1 The Basic Problem

The basic principle behind model-driven process-execution-systems (work-flow management systems) is that they use a model (most often a graph/diagram) and load it into an interpreter machine, thus forming an instance of the model. This instance, model and interpreter together, can be considered as a state machine which can be executed, or run through, until it has finished.

One such model language is the Parallel Activity Specification Schema (PASS) introduced by Albert Fleischmann in [Fle94]. In [Bör11] Egon Börger has presented an ASM specification for a PASS interpreter for single SBDs.

A great challenge for model based process execution system is that the models may need to change in order to cope with change requirements in the real-life processes that they are representing and supporting. With short lived process instances that is no problem. There are cases, though, where the execution of a process instance can take weeks or months. During such duration there usually is a big chance for circumstances to arise that in turn require changing at least parts of the process model ad-hoc in order to satisfy the new needs without restarting whole process instances. The problem is not new and described, e.g. in [DR09], and research into formal requirements for such mechanism dates at least back to [RD98].

An – admittedly brief – overview of the research has led to the impression that such mechanisms may although not be explicitly be applicable to PASS models and their separated-graph nature. Furthermore, the here proposed mechanism can also be used to handle the execution of multi-behavior subjects.

The basic idea to allow for an update mechanism is to incorporate a model-exchange-mechanism into the model interpreter machine. It is assumed that a mechanism exists that guaranties validity of all parts of a model in the current context and that supporting tools can be realized in a way that a multi-behavior model is valid or more specifically any diagram $D*$ (e.g. any **Extension Behavior** , **Guard Behavior** , or **Macro-Behavior**) will be a valid extension of D (e.g. a **Base Behavior**) in a given currently running process context. A function $validInCurrentAmb(D*, D)$ will be the placeholder.

The question up to discussion is whether the arbitrator pattern can be used for the task of realizing multi-behavior execution and the execution of ad-hoc extension to process models?

5.8.2 The arbitrator pattern

The arbitrator pattern stems from the field of robotics and was introduced by R.C. Arkin in [Ark98] to program LEGO Mindstorm robots to interact with their not predefined environment. It allows for fast and effective programming of independent robots, but was advised against for use anywhere else, but robotic. Its basic principle assumes a robot with input and output equipment. Instead of programming a single large complex program to control the machine there is one arbitrator deciding which of many smaller behavior programs (short "behaviors") is currently to be executed. These behaviors may contain only simple

instructions like "move forward". Which behavior is currently controlling the robot is determined by a dynamically reevaluated priority that is defined based on the sensor inputs for each behavior. As soon as external events (e.g. the robot hitting a wall) require a change, the priority is shifted and the arbitrator executes a different behavior. Behaviors can be added as required given that priority-computing-functions are included.

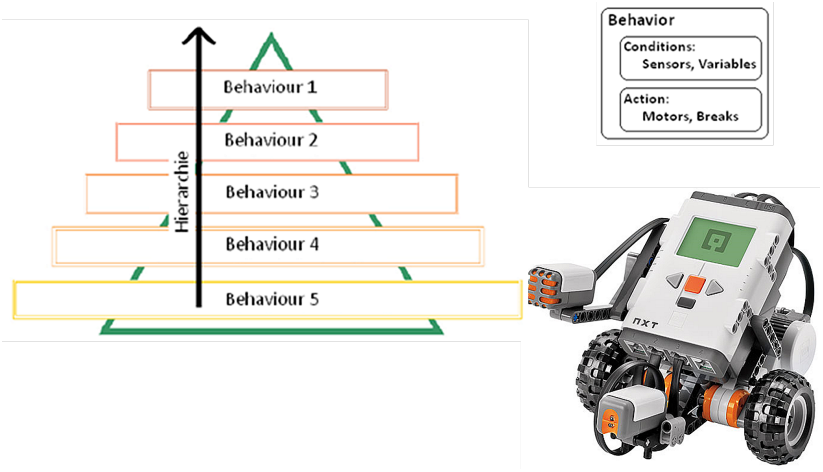


Figure 5.42: Principle Arbitrator Pattern as Intended by [Ark98]

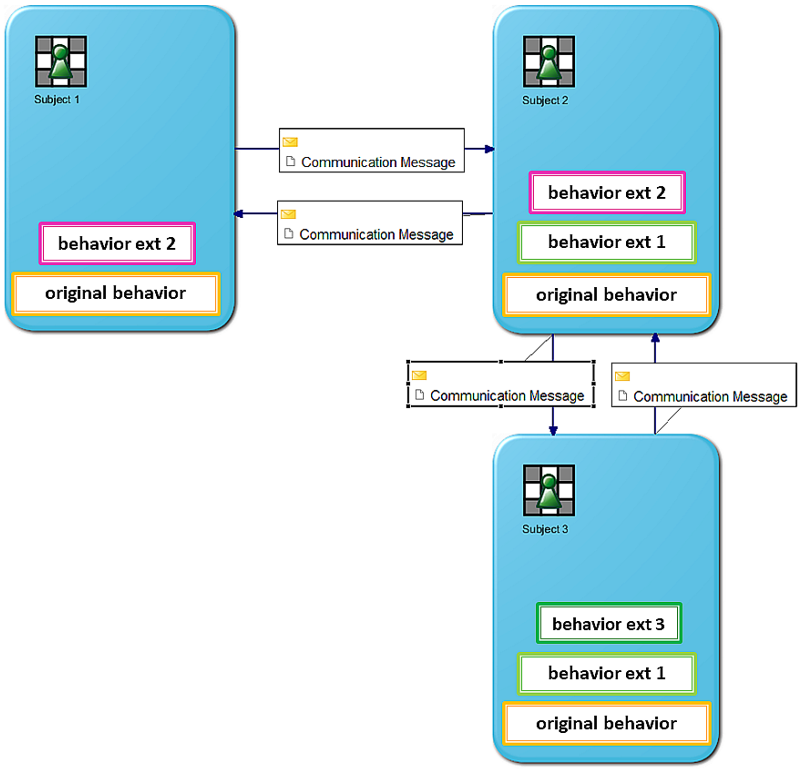


Figure 5.43: Multi-Behavior Subjects

5.8.3 The arbitrator pattern for PASS execution

The idea is to use the Arbitrator Pattern as basis for a mechanism to allow dynamic adaption and multi-behavior execution within an instance in work flow management system for PASS. Instead of Motors and Sensors, in the context of PASS, a Subject interacts with its environment via the reception and sending of Messages which can be directed to other subjects (the environment). Furthermore, internal inputs (user choices or other computations that determine actions of a Subject) can, or rather should affect the priorities of behaviors (i.e.: can determine which behavior is executed). The equivalent to the sensors of a robot here is a subject's 'message In-Box'. And instead of controlling motors, here the arbitrator grants a Subject-Behavior the right to access the message facilities – to receive and send messages. Consequently, the core concept here is not to handle a subject as single SBD-interpreter-machine, but rather as multiple interpreter machines encapsulated in an arbitrator-controlled-machine which can grant control rights for the unit/subject. Towards the outside a subject is in principle a single unit with its Message In-Box and outgoing messages. The original $BEHAVIOR(subj, state)$ ASM defined by Börger in [Bör11] needs to be adapted in order to really fit into the concept presented here. For now it is assumed that the definition will work. Under normal circumstances i there should be only one behavior container, containing a $BEHAVIORD(subj, state) - ASM$. For the arbitrator pattern to work, it is assumed to be encapsulated in a container which has a priority and execution functionality:

$BEHAVIOR\_INTERPRETER\_CONTAINER(D, subj, state, containerID) =$
seq
if $couldTakeControl(D, subj, state)$ **then**
$updatePriorityForArbitrator(subj, thisContainer)$
if $hasControl(subj, containerID)$ **then**
$BEHAVIORD(subj, state)$

This definition should express that, when executed, the behavior container checks whether it $couldTakeControl$ of the subject or not and updates the priorities. Based on that priority list the arbitrating machine will grant the access right via the take-Control command that should evaluate as true for the empowered $BEHAVIOR\_INTERPRETER\_CONTAINER$ machine. Upon a special change request – e.g. a special message (or event) outside the process context containing a new model ($behaviorExtensionArrived$) – the $ARBITRATOR(subj, context)$ can initialize and/or start a new interpreter machine based on the received model D to take control of the subject. The condition of course, and the need for further research, is that the new model data $isValidForContext(newestBehaviorDiagram(subj), proc$ in order to fit logically into the current process context which is given by the model the original behavior-machine is based on. The initialized machines need to be traced/collected in a location of $activeBehaviours(subj)$.The source of such a special message for now is assumed to be an administrator outside the process context. More elaborate or sophisticated mechanisms are imaginable. By default a newer model simply has a higher priority for execution. Further rules to determine priority will be needed.

An attempt to specify the described mechanism with the means of ASMs is given here:

$ARBITRATOR(subj, processContext)$
seq
if $behaviorExtensionArrived(subj, processContext, D*)$ **then**
if $isValidForContext(newestBehaviorDiagram(subj), processContext)$ **then**
$initializeNewBehaviorInterpreterContainer(newestBehaviorDiagram(subj))$
else forall i in activeBehaviours(subj) **do**
$hasControl(subj, i) := false$
$BEHAVIOR\_INTERPRETER\_CONTAINER(D, subj, state)$
choose j in $activeBehaviours(subj)$ **where** $priority(j) > priority(x)x! = j$
hasControl(subj, j) := true

This machine should execute with a certain frequency, repeating the cycle and being aware of new behaviors, reevaluating priorities and (re-)granting control to a behavior continuously. In the case of the original robotics concept, behavior changes can occur in the span of milliseconds depending on the power and speed of the controlling computer. In a business process context, a behavior change may not need the strict real-time requirement since a change might not occur at all or only a few times in the span of, e.g., a minuet.

5.8.4 Final Thoughts

The proposed mechanism is not very complicated. In robotics this simple approach allows to construct complex behavior out of simple elements. Our future research will be aimed at investigating whether the application of this concept here can be useful and to find possible drawbacks (e.g. validation concerns) and challenges that would need to be addressed before actual applying this concept. First of all being the definition of valid model extensions mechanism/validator rules for PASS, followed by issues like the priority determination and the question about frequency of priority updates and actual behavior changes among other details needed to actually build a prototypical implementation as the high goal.

APPENDIX A

Classes and Properties of the PASS Ontology

A.1 All Classes (95)

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

- ReceiveFunction ; SRN: 008
  Specifications/descriptions for Receive-Functions describe in detail what the subject carrier is supposed to do in a state.
  DefaultFunctionReceive1_EnvoironmentChoice : present the surrounding execution environment with the given exit choices/conditions currently available depending on the current state of the subjects in-box. Waiting and not executing the receive action is an option.
  DefaultFunctionReceive2_AutoReceiveEarliest: automatically execute the according activity with the highest priority as soon as possible. In contrast to DefaultFunctionReceive1, it is not an option to prolong the reception and wait e.g. for another message.
  - SendFunction ; SRN: 009
    Comments have to be added
- DoFunction ; SRN: 010
  Specifications or descriptions for Do-Functions describe in detail what the subject carrier is supposed to do in an according state. The default DoFunction
  1: present the surrounding execution environment with the given exit choices/-conditions and receive choice of one exit option –> define its Condition to be fulfilled in order to go to the next according state. The default DoFunction
  2: execute automatic rule evaluation (see DoTransitionCondition - ToDo) More specialized Do-Function Specifications may contain Data mappings denoting what of a subjects internal local Data can and should be:
  a) read: in order to simply see it or in order to send it of to an external function (e.g. a web service)
  b) write: in order to write incoming Data from e.g. a web Service or user input, to the local data fault

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- SubjectBaseBehavior ; SRN: 123
  The standard behavior model type

A.2 Object Properties (42)

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

Property name		Domain-Range	Comments	Reference
hasDataDefinition	Domain: Range:	 DataObjectDefinition		210
hasDataMapping-Function	Domain: Range:	state, SendTransition, ReceiveTransition DataMappingFunction		211
hasDataType	Domain: Range:	PayloadDescription or DataObjectDefinition DataTypeDefinition		212
hasEndState	Domain: Range:	SubjectBehavior or ChoiceSegmentPath State, not SendState		213
hasFunction-Specification	Domain: Range:	State FunctionSpecification		214
hasHandlingStrategy	Domain: Range:	InputPoolConstraint InputPoolContstraint-HandlingStrategy		215
hasIncomingMessage-Exchange	Domain: Range:	Subject MessageExchange		216
hasIncomingTransition	Domain: Range:	State Transition		217
hasInitialState	Domain: Range:	SubjectBehavior or ChoiceSegmentPath State		218
hasInputPoolConstraint	Domain: Range:	Subject InputPoolConstraint		219
hasKeyValuePair	Domain: Range:			220
hasMessageExchange	Domain: Range:	Subject	Generic super class for the 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 – the state the subject is currently in as defined by a **State** in the model	*SID_state*	Execution concept – no model representation, Not to be confused by a model "state" in an SBD Diagram. State in the SBD diagram define possible SID_States.
SubjectBehavior – under the assumption that it is complete and sound.	*D*	A Diagram that is a completely connected SBD
State	*node*	A specific element of diagram D - Every node 1:1 to state
State	*state*	The current active state of a diagram determined by the nodes of Diagram D
InitialStateOfBehavior, **EndState**	*initial state,* *end state*	The interpreter expects and SBD Graph D to contain exactly one initial (start) state and at least one end state.
Transition	*edge / outEdge*	"Passive Element" of an edge in an SBD-graph
TransitionConditionn	*ExitCondition*	Static Concept that represents a Data condition
Execution Concept – ID of a Subject Carrier responsible possible multiple Instances of according to specific **SubjectBehavior**	*subj*	Identifier for a specific Subject Carrier that may be responsible for multiple Subjects
Represented in the model with **InterfaceSubject**	*ExternalSubject*	A representation of a service execution entity outside of the boundaries of the interpreter (The PASS-OWL Standardization community decided on the new Term of Interface Subject to replace the often-misleading older term of External Subject)
SubjectBehavior or rather **SubjectBaseBehavior** as MacroBehaviors and GuardBehaviors are not covered by Börger	*subject-SBD* / $SBDsubject_{subject}$	Names for completely connected graphs / diagrams representing SBDs
Object Property: ***hasFunctionSpecification*** (linking **State**, and **FunctionSpecification** –>(**State** ***hasFunctionSpecification*** **FunctionSpecification**)	*service(state)* / *service(node)*	Rule/Function that reads/returns the service of function of a given state/node

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

B.2 Main Execution/Interpreting Rules

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

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

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

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

Table B.2: Main Execution/Interpreting Rules

B.3 FUNCTIONS

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

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

Table B.3: Derived Functions

B.4 EXTENDED CONCEPTS – REFINEMENTS FOR THE SEMANTICS OF CORE ACTIONS

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

Table B.4: Refinements places

B.5 INPUT POOL HANDLING

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

Table B.5: Input Pool Handling

B.6 OTHER FUNCTIONS

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

Table B.6: Other Functions

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

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

Table B.7: Other Functions

APPENDIX C

CoreASM PASS Reference Implementation

The CoreASM PASS Reference Implementation is based on theoretical works by Egon Börger [Bör11] and Manuel Bandmann [Ban14], which predate the OWL description, and incorporates concepts, that are not (yet) part of the agreed PASS Standard and OWL description. The first goal of the reference implementation was to enable the interactive process model validation based on formal PASS execution semantics, i.e. to demonstrate that the specification is actually executable and sound. In order to reach that goal not all known PASS concepts could be implemented at once, for example only one InputPoolConstraintStrategy is currently implemented. Section C.2 describes the conceptual differences and missing elements to the OWL description.

The bigger part of this appendix lists relevant Abstract State Machines (ASM) [BS03] rules in the CoreASM dialect in an edited form, see section C.3 for the applied editorial changes.

The full CoreASM PASS Reference Implementation is publicly available at [Wol20]. The following section C.1 gives an overview of the main elements of the reference implementation.

C.1 Architecture

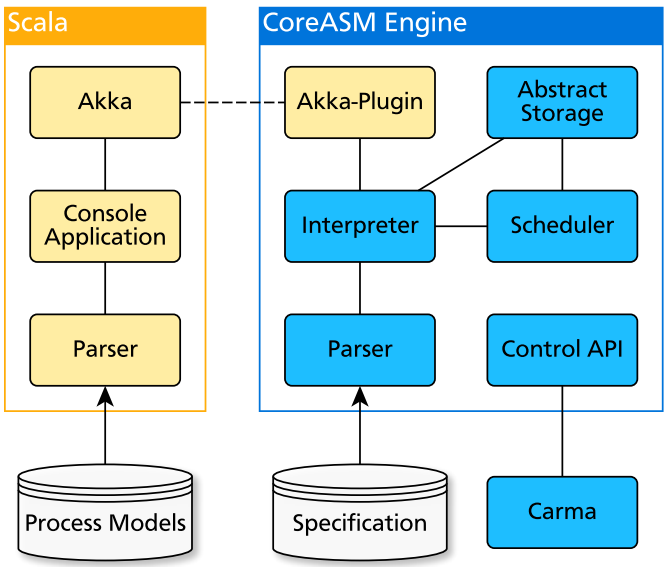


Figure C.1: Architecture of the reference implementation

The overall architecture of the CoreASM PASS Reference Implementation 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. For stability and performance improvements the fork [Wol19] of the CoreASM framework [FG11] is used. 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 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.

C.2 Conceptual Differences to the OWL Description

The most important differences are listed in Table C.1: while the corresponding OWL Element Description offers both synchronous and asynchronous communication, different InputPool limits and handling strategies this reference implementation addresses only asynchronous communication with a blocking strategy. The End State property is not supported, but instead a Terminate Function is used. 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. [id=AW]Are Macro Exit Parameters no longer part of PASS (see table C.2)? I think they were present in the original Fleischmann book, the Börger asm interpreter and the 2012 book. 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	Corresponding OWL Element 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	Automatically when an additional message arrives on a Subject that is *proper terminated*.	*Absent*.
Proper Termination	When a Subject terminates and its InputPool is empty it is *proper terminated* and can be restarted.	*Absent*.
End State	Continuing the Subject behavior is not supported. Implemented as **TerminateFunction**, which determines *proper termination* and enables *subject restart*. It must not have any outgoing transitions.	An **EndState** on a **DoState** or **ReceiveState** with an outgoing transition allows to continue the Subject behavior.
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 Corresponding OWL Element 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.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.

Concept	Reference Implementation	Corresponding OWL Element 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

```
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.

Finally, the CoreASM specification files are annotated with numerous debug statements and internal data validation checks, that are used for continuous integration testing, which have also been removed in order to preserve the printed space.

Concept	Reference Implementation	Corresponding OWL Element 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

Concept	Reference Implementation	Corresponding OWL Element 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.4 Basic Definitions

```
function channelFor : Agents -> LIST

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

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

Listing 33: channelFor

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

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

Listing 34: activeStates

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

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

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

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

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

Listing 35: nextPI

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

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

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

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

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

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

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

Listing 36: initializedState

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

Listing 37: shouldTimeout

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

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

Listing 38: variable

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

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

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

Listing 39: macroTerminationResult

C.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 PerformTerminate 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 PerformTerminate w/o ASM step
      "terminate"      : 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)
    "terminate"      : PerformTerminate(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 TERMINATE FUNCTION

```
rule PerformTerminate(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: PerformTerminate

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((max > 0 and hasSizeWithin(predefAgents, min, max) = true) or
          (max = 0 and hasMinimalSize(predefAgents, min) = 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

[AGT+17] M. Ackerman, S. Goggins, Herrmann Th., M. Prilla, and Ch. Stary. *Designing healthcare that works*. Academic Press, Amsterdam, 2017.

[Ark98] Ronald Arkin. *Behavior-based robotics*. The MIT Press, 1998.

[ASM09] A-Sharp and P. McDermott. *Workflow Modeling*. Artech House, 2009.

[AWC14] Fleischmann A., Schmidt Werner, and Stary Christian. Subject oriented project management. In *SEAA '14: Proceedings of the 2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*. IEEE Computer Society, 2014.

[Ban14] Manuel Bandmann. Spezifikation einer Ausführungs- und Verifikationseinheit mit Abstract State Machines für die Subject-Oriented Business Process Management Modellierungsnotation. Diplomarbeit, TU Darmstadt, 2014.

[Ber11] M. Berghaus. *Luhmann leicht gemacht*. Böhlau Verlag, 2011.

[BMNZ14] F. Bonomi, R. Milito, P. Natarajan, and J. Zhu. Fog computing: A platform for internet of things and analytics. In *Big Data and Internet of Things: A Roadmap for Smart Environments*, pages 169–186. Springer International Publishing, 2014.

[Boe12] E. Boerger. Approaches to modeling business processes: a critical analysis of bpmn, workflow patterns and yawl. *Journal of Software and Systems Modeling*, 11:305–318, 2012.

[Bör11] Egon Börger. A Subject-Oriented Interpreter Model for S-BPM. *Appendix in: A. Fleischmann, W. Schmidt, C. Stary, S. Obermeier, E. Börger: Subjektorientiertes Prozessmanagement, Hanser-Verlag, München*, 2011.

[BR18] E. Börger and A. Raschke. *Modeling Companion for Software Practitioners*. Springer Verlag, 2018.

[BS97] W. Becker and N. Sahl. Erfüllbarkeit bedeutsamer rechenzwecke durch die prozesskostenrechnung – dargestellt am beispiel der wirtschaftlichkeitskontrolle in administrative

leistungsbereichen. *Bamberger betriebswirtschaftliche Beiträge Nr. 117*, 1997.

[BS03] E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag, 2003.

[CGK19] L. Golab C. Gorenflo, S. Lee and S. Keshav. Fastfabric: Scaling hyperledger fabric to 20,000. *Online Available: arXiv:1901.00910v2*, Accessed 14 February 2019.

[CK88] R. Cooper and R.S. Kaplan. Measure costs right: Make the right decisions. *Harvard Business Review*, pages 96–103, 1988.

[Con68] M. E. Conway. How do committees invent? In *Datamation*, page 28–31, 1968.

[DCL+17] T. A. Dinh, G. Chen, R. Liu, B. C. Ooi, J. Wang, and K. L. Ta. Blockbench: A framework for analyzing private blockchains. In *Proceedings of the 2017 ACM International Conference on Management, SIGMOD '17*. SIGMOD, 2017.

[DEG03] Avison D. E. and Fitzgerald G. *Information Systems Development: Methodologies, Techniques and Tools*. 3rd ed. McGraw-Hill, 2003.

[Din06] T. Dinter, B.and Bucher. Business performance management. In P. Chamoni and P. Gluchowski, editors, *Analytische Informationssysteme*, pages 23–50. Springer Verlag, 2006.

[DR09] Peter Dadam and Manfred Reichert. The adept project: A decade of research and development for robust and flexible process support. Technical Report UIB-2009-01, University of Ulm, Ulm, January 2009.

[Dre17] D. Drecher. *Blockchain Basics*. Apress Inc.USA, 2017.

[DRODRDTRC12] A. Del-Río-Ortega, M. De Reyna, A. Durán Toro, and A. Ruiz-Cortés. Defining process performance indicators by using templates and patterns. In A. Barros, A. Gal, and E. Kindler, editors, *BPM 2012*, LNCS 7481, pages 223–228. Springer Verlag, 2012.

[ea01] J. Snabe et. al. *Business Process Management, The SAP Roadmap*. Springer Verlag, 4.edition edition, 2001.

[ECF+18] C. Esposito, F. Castiglione, A.and Palmieri, M. Ficco, C. Dobre, G. V. Iordache, and F. Pop. Event-based sensor data exchange and fusion in the internet of things environments. *Journal of Parallel and Distributed Computing*, 2018.

[EF19] M Elstermann and A. Fleischmann. Modeling complex process systems with subject oriebted means. In S. Betz, M. Elstermann, and M Lederer, editors, *S-BPM ONE 2019, 11th International Conference on Subject Oriented Business Process*

Management, ICPC published by ACM Digital Library. Association of Computing Machinery (ACM), 2019.

[Els17] M. Elstermann. Proposal for using semantic technologies as a means to store and exchange subject-oriented process models. *S-BPM ONE 2017 - Darmstadt, Germany — March 30 - 31*, 2017.

[Els19] M. Elstermannn. *Executing Strategic Product Planning - A Subject-Oriented Analysis and New Referential Process Model for IT-Tool Support and Agile Execution of Strategic Product Planning*. KIT Scientific Publishing, 2019.

[EN11] O. Etzion and P. Niblett. *Event Processing in Action*. Stamford, 2011.

[ESF12] M. Elstermann, D. Seese, and A. Fleischmann. Using the arbitrator pattern for dynamic process instance extension in a work-flow management system. *Abstract State Machines, Alloy, B, VDM, and Z*, pages 323–327, 2012.

[et.11] Albert Fleischmann et.al. *Subjektorientiertes Prozessmanagement*. Hanser Verlag, 2011.

[FBE+17] A. Fleischmann, S. Borgert, M. Elstermann, F. Krenn, and R. Singer. An overview to s-bpm oriented tool suites. In *Proceedings of the 9th International Conference on Subject-oriented Business Process Management*, S-BPM ONE. ACM, 2017.

[FG11] Roozbah Farahbod and Uwe Gläser. The coreasm modeling framework. *Software: Practice and Experience*, 41(2), February 2011.

[FJMM12] J.-P. Friedenstab, C. Janiesch, M. Matzner, and O. Müller. Extending bpmn for business activity monitoring. In *45th Hawaii International Conference on System Sciences*, pages 4158–4167, 2012.

[Fle94] Albert Fleischmann. *Distributed Systems - Software Design and Implementation*. Springer Verlag, Berlin, 1994.

[Fle13] A. Fleischmann. Subject-phase model based process specifications. In H. Fischer and J. Schneeberger, editors, *Proceedings of the 5th International Conference*, Communications in Computer and Information Sciences (CCIS). Springer Verlag, 2013.

[Fou19] The Linux Foundation. Hyperledger. *https://www.hyperledger.org/*, Accessed 14 February 2019.

[FS11] A. Fleischmann and C. Stary. Whom to talk to? a stakeholder perspective on business process management. *Universal Access in the Information Society*, 2011.

[FS19] A. Fleischmann and C. Stary. Dependable data sharing in dynamic iot-systems - subject-oriented process design, complex event processing, and blockchains. In S. Betz, M. Elstermann, and M Lederer, editors, *S-BPM ONE 2019, 11th International Conference on Subject Oriented Business Process Management*, ICPC published by ACM Digital Library. Association of Computing Machinery (ACM), 2019.

[FSS+12a] Albert Fleischmann, Werner Schmidt, Christian Stary, Stefan Obermeier, and Egon Boerger. *Subject-Oriented Business Process Management*. Springer Verlag, Berlin, 2012.

[FSS+12b] Albert Fleischmann, Werner Schmidt, Christian Stary, Stefan Obermeier, and Egon Boerger. *Subject-Oriented Business Process Management*. Springer Verlag, Berlin, 2012.

[GDG16] H. Gupta, A. V. Dastjerdi, and R. Ghosh, S. K.and Buyya. ifogsim: A toolkit for modeling and simulation of resource management techniques in internet of things, edge and fog computing environments. In *arXiv preprint arXiv:1606.02007*, 2016.

[GJH09] Marx Gomez, Junker J., and S. H., Odebrecht. *IT-Controlling*. Erich Schmidt Verlag, Berlin, 2009.

[Gro15] Norbert Gronau. Industrie 4.0. *Enzyklopedie der Wirtschaftsinformatik*, 2015.

[Gru02] V. Gruhn. *Process-centered software engineering environments, a brief history and future challenges*, pages 363–382. Springer, 2002.

[Hab81] J. Habermas. *Theory of Communicative Action Volume 1, Volume 2*. Suhrkamp Paperback Science, 1981.

[Hes05] H Hess. Von der unternehmensstrategie zur prozessperformance was kommt nach business intelligence? In A. Scheer A.-W.and Jost, W.and Hess H.and Kronz, editor, *Corporate Performance Management*. Springer Verlag, 2005.

[HFP03] Smith H. F. P. *Business process management—The third Wave*. Meghan-Kiffer Press, 2003.

[HM89] P. Horvath and R. Mayer. Prozesskostenrechnung – der neue weg zu mehr kostentransparenz und wirkungsvolleren unternehmensstrategien. *Controlling*, pages 214–219, 1989.

[Hoa85] A. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[HP05] Horvath and Partner, editors. *Prozessmanagement umsetzen*. Schäfer, Pöschel, 2005.

[HXW13] D. Huang, T. Xing, and H. Wu. Mobile cloud computing service models: A user-centric approach. In *IEEE Netw.*, page 6–11. IEEE, 2013.

[Jam11] M. Ed. Jamshidi. System of systems engineering: innovations for the twenty-first century. In *Big Data and Internet of Things: A Roadmap for Smart Environments*. John Wiley and Sons, 2011.

[JH13] Ludewig J. and Lichter H. *Software Engineering*. 3. Edition, d-punkt Verlag, 2013.

[JMM11a] C. Janiesch, M. Matzner, and O. Müller. A blueprint for event-driven business activity management. In S. Rinderle, F. Toumani, and K. Wolf, editors, *9th International Conference on Business Process Management (BPM)*, LNCS 6896. Springer Verlag, 2011.

[JMM11b] C. Janiesch, M. Matzner, and O. Müller. A blueprint for event-driven business activity management. In S. Rinderle, F. Toumani, , and K. Wolf, editors, *9th International Conference on Business Process Management (BPM)*, pages 17–28. Springer, 2011.

[JONB04] Coplien J. O. and Harrison N. B. *Organizational Patterns of Agile Software Development*. Prentice Hall International, 2004.

[JS98] Putman J. and Strong S. A federated virtual enterprise (ve) of partners, creating a federated ve of systems. In *IEEE Proceedings of Compsac*, 1998.

[Kee76] E. L. Keenan. Towards a universal definition of 'subject'. In Li Charles, N., editor, *Subject and topic*. Academic Press New York, 1976.

[KS16] F. Krenn and C. Stary. Exploring the potential of dynamic perspective taking on business processes. *Complex Systems Informatics and Modeling Quarterly (8)*, pages 15–27, 2016.

[KSW16] F. Krenn, C. Stary, and D. Wachholder. Stakeholder-centered process implementation: Assessing s-bpm tool support. In *Proceedings of the 9th International Conference on S-BPM (S-BPM-One)*, page 2. ACM, 2016.

[Kue09] M. Kuetz. *Kennzahlen in der IT: Werkzeuge für Controlling und Management*. dpunkt, Heidelberg, 2009.

[LLX09] Chengfei L., Qing L., and Z. Xiahui. Challenges and opportunities in collaborative business process management: Overview of recent advances and introduction to the special issue. In *Information Systems Frontiers*, pages 201–209, 2009.

[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.

[Luh84] N. Luhmann. *Social Systems*. Suhrkamp Verlag, 1984.

[Mei13] J. Meißner. Cyberphysische Produktionssysteme. *Productivity Management*, 18(1):21–24, 2013.

[MFR+10] Nils Meyer, Thomas Feiner, Markus Radmayr, Dominik Blei, and Albert Fleischnann. Dynamic creation and execu-tion of cross organisational business processes the jcpex! approach.

In A. Fleischmann, W. Schmidt, R. Singer, and D. Seese, editors, *S-BPM ONE: Setting the Stage for Subject-Oriented Business Process Management: First International Workshop*, number 138 in Communications in Computer and Information Sience. Springer Verlag, 2010.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Mil99] R. Milner. *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge University Press, 1999.

[MV85] J.G. Miller and T.E. Vollmann. The hidden factory. *Harvard Business Review*, pages 142–150, 1985.

[OMG18] OMG. *Business Process Model and Notation (BPMN)*. http://www.omg.org/spec/BPMN2.0, last access June 2018.

[Peh12] A Pehlke. *Aufbau eines Business Activity Monitoring für den subjektorientierten Ansatz des Business Process Managements*. Bachelor Thesis, University of Applied Sciences Ingolstadt, 2012.

[RD98] Manfred Reichert and Peter Dadam. Adeptflex-supporting dynamic changes of workflows without losing control. *Journal of Intelligent Information Systems, Special Issue on Workflow Management Systems*, 10(2):93–129, March 1998.

[Röm15] M. Römpp. *Habermas leicht gemacht*. Böhlau Verlag, 2015.

[RRF08] Ramsin R. and Paige R. F. Process-centered review of object oriented software development methodologies. In *ACM Computing Surveys*, volume Volume-40, Number 1, 2008.

[SAO09] W Schmidt, Fleischmann A., and Gilbert O. Subjektorientiertes geschaeftsprozessmanagement. *HMD Praxis der Wirtschaftsinformatik*, Heft 266, 2009.

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

[Sch01] A. W. Scheer. *ARIS-Vom Geschäftsprozess zum Anwendungssystem*. Springer Verlag, 4.edition edition, 2001.

[Sch13] W Schmidt. Business activity monitoring. In P. Rausch, A. Sheta, and A. Ayesh, editors, *Business Intelligence and Performance Management Theory, Systems and Industrial Applications*. Springer UK, 2013.

[SDW+15] Y. Shi, G. Ding, H. Wang, H. E. Roman, and S. Lu. The fog computing service for healthcare. In *2nd International Symposium on Future Information and Communication Technologies for Ubiquitous HealthCare (Ubi-HealthTech)*, pages 1–5. IEEE, 2015.

[SF13] W. Schmidt and A. Fleischmann. Business process monitoring with s-bpm. In H. Fischer and J. Schneeberger, editors, *Proceedings of the 5th International Conference S-BPM ONE 2013, Computer and Information Sciences (CCIS), No. 360*. Springer, 2013.

[SFG09] W. Schmidt, A. Fleischmann, and O. Gilbert. Subjektorientiertes geschäftsprozessmanagement. *HMD – Praxis der Wirtschaftsinformatik*, pages 52–62, 2009.

[SFI13] M. H. Syed, E. B. Fernandez, and M. Ilyas. A pattern for fog computing. In *Proceedings of the 10th Travelling Conference on Pattern Languages of Programs*, page 13. ACM, 2013.

[SFS18] C. Stary, A. Fleischmann, and W. Schmidt. Subject-oriented fog computing: Enabling stakeholder participation in development. In *Proceedings of the 4th IEEE World Forum on Internet of Things (WF-IoT), Singapure*. IEEE Xplore Digital Library, DOI 10.1109/WF-IoT.2018.8355167, 2018.

[SN11] D. Slama and R. Nelius. *Enterprise BPM*. dpunktVerlag, 2011.

[SS10] H. Schmelzer and W. Sesselmann. *Geschäftsprozessmanagement in der Praxis*. Hanser Verlag, 2010.

[vBM10] J. vom Brocke and M.Rosemann, editors. *Handbook on Business Process Management*, volume 2. Springer Verlag, 2010.

[VRM14a] L.M. Vaquero and L. Rodero-Merino. Finding your way in the fog: Towards a comprehensive definition of fog computing. In *ACM SIGCOMM Computer Communication Review*, pages 27–32. ACM, 2014.

[VRM14b] L.M. Vaquero and L. Rodero-Merino. Policy-driven security management for fog computing: Preliminary framework and a case study. In *IEEE 15th International Conference on Information Reuse and Integration (IRI)*, pages 16–23. IEEE, 2014.

[Wal19] J. Waldo. A hitchhiker's guide to the blockchain universe. *Communications of the ACM*, 2019.

[WBH19a] A Wolski, S. Borgert, and L. Heuser. A coreasm based reference implementation for subject oriented business process management execution semantics. In S. Betz, M. Elstermann, and M Lederer, editors, *S-BPM ONE 2019, 11th International Conference on Subject Oriented Business Process Management*, ICPC published by ACM Digital Library. Association of Computing Machinery (ACM), 2019.

[WBH19b] A Wolski, S. Borgert, and L. Heuser. An extended subject-oriented business process management execution semantics. In S. Betz, M. Elstermann, and M Lederer, editors, *S-BPM ONE 2019, 11th International Conference on Subject Oriented*

Business Process Management, ICPC published by ACM Digital Library. Association of Computing Machinery (ACM), 2019.

[Wol19] André Wolski. Fork of the CoreASM Framework. `https://github.com/Locke/coreasm.core`, 2019. [Online; accessed 2020-11-08].

[Wol20] André Wolski. CoreASM-based PASS interpreter. `https://github.com/I2PM/asm-pass-interpreter`, 2020. [Online; accessed 2020-11-08].

[www] www.w3.org. Web ontology language (owl).

[www04] www.w3.org. Xml schema. `https://www.w3.org/standards/xml/schema`, 2004. Online; access 9th November 2020.

[www15a] www.w3.org. Resource description framework. `https://www.w3.org/RDF/`, 2015. [Online; accessed 2016-12-12].

[www15b] www.w3.org. Resource description framework schema. `https://www.w3.org/2001/sw/wiki/RDFS`, 2015. [Online; accessed 2016-12-12].

[www16] www.w3.org. The web ontology language. `https://www.w3.org/TR/owl2-overview`, 2016. [Online; accessed 2016-12-12].

[XWS19] X. Xu, I. Weber, and M. Staples. *Architecture for Blockchain Applications*. Springer Nature, 2019.

[YL15] S. Yi and Q. Li, C.and Li. A survey of fog computing: concepts, applications and issues. In *Proceedings of the 2015 Workshop on Mobile Big Data*, pages 37–42. ACM, 2015.

[ZSF13] C. Zehbold, W. Schmidt, and A. Fleischmann. Activity-based costing for s-bpm. In H. Fischer and J. Schneeberger, editors, *Proceedings of the 5th International Conference*, Communications in Computer and Information Sciences (CCIS). Springer Verlag, 2013.

Todo list