MEMO Organisation Modelling Language (2): focus on business processes

Frank, Ulrich

In: ICB Research Reports - Forschungsberichte des ICB / 2011

This text is provided by DuEPublico, the central repository of the University Duisburg-Essen.

This version of the e-publication may differ from a potential published print or online version.

DOI: https://doi.org/10.17185/duepublico/47065

URN: urn:nbn:de:hbz:464-20180918-065639-7

Link: https://duepublico.uni-duisburg-essen.de/servlets/DocumentServlet?id=47065

License:
As long as not stated otherwise within the content, all rights are reserved by the authors / publishers of the work. Usage only with permission, except applicable rules of german copyright law.

Source: ICB-Research Report No. 49, December 2011

Ulrich Frank

MEMO Organisation Modelling Language (2): Focus on Business Processes

49

ICB-RESEARCH REPORT

ICB-Research Report No. 49
December 2011

Die Forschungsberichte des Instituts für Informatik und Wirtschaftsinformatik dienen der Darstellung vorläufiger Ergebnisse, die i. d. R. noch für spätere Veröffentlichungen überarbeitet werden. Die Autoren sind deshalb für kritische Hinweise dankbar.

The ICB Research Reports comprise preliminary results which will usually be revised for subsequent publications. Critical comments would be appreciated by the authors.

Alle Rechte vorbehalten. Insbesondere die der Übersetzung, des Nachdruckes, des Vortrags, der Entnahme von Abbildungen und Tabellen – auch bei nur auszugsweiser Verwertung.

All rights reserved. No part of this report may be reproduced by any means, or translated.

Authors' Address:

Ulrich Frank

Lehrstuhl für Wirtschaftsinformatik
und Unternehmensmodellierung

Institut für Informatik und
Wirtschaftsinformatik (ICB)
Universität Duisburg-Essen
Universitätsstr. 9
D-45141 Essen

ulrich.frank@uni-due.de

ICB Research Reports

Edited by:

Prof. Dr. Heimo Adelsberger
Prof. Dr. Peter Chamoni
Prof. Dr. Frank Dorloff
Prof. Dr. Klaus Echtle
Prof. Dr. Stefan Eicker
Prof. Dr. Ulrich Frank
Prof. Dr. Michael Goedicke
Prof. Dr. Volker Gruhn
PD Dr. Christina Klüver
Prof. Dr. Tobias Kollmann
Prof. Dr. Bruno Müller-Clostermann
Prof. Dr. Klaus Pohl
Prof. Dr. Erwin P. Rathgeb
Prof. Dr. Enrico Rukzio
Prof. Dr. Albrecht Schmidt
Prof. Dr. Rainer Unland
Prof. Dr. Stephan Zelewski

Contact:

Institut für Informatik und
Wirtschaftsinformatik (ICB)
Universität Duisburg-Essen
Universitätsstr. 9
45141 Essen

Tel.: 0201-183-4041
Fax: 0201-183-4011
Email: icb@uni-duisburg-essen.de

ISSN 1860-2770 (Print)
ISSN 1866-5101 (Online)

Abstract

An enterprise model comprises various abstractions of an enterprise that represent both information systems and the surrounding action systems. These different models are integrated in order to avoid redundant work and to contribute to a tight and consistent integration of action systems and information systems. For this purpose, the method MEMO (Multi-Perspective Enterprise Modelling) features a family of modelling languages, each of which is aimed at representing specific perspectives and aspects of an enterprise. Within the MEMO languages, the Organisation Modelling Language (MEMO OrgML) is of outstanding relevance. Its specification is distributed on two reports. A previous report focussed on the specification of the organisation structure. This report presents the part of the language specification that allows for modelling business processes. It builds on a further previous report that presents the results of an extensive requirements analysis. The report starts with an analysis of basic concepts and a discussion of peculiarities related to the specification of a process modelling language. Subsequently, the language specification is presented. It comprises a meta model with supplementary constraints, detailed comments on core language concepts and a dictionary of the notational symbols in two variants. Finally, the use of the language is demonstrated with examples.

Table of Contents

TYPOGRAPHICAL CONVENTIONS VIII

ACKNOWLEDGEMENTS IX

1 INTRODUCTION 1

2 BASIC CONCEPTS 4

2.1 PROCESS 5

2.1.1 Business Process 5

2.1.2 Subprocess 6

2.2 EVENT 7

2.2.1 A Generic Taxonomy of Events 8

2.2.2 Conceptualiation of Events 10

2.3 EXCEPTION 14

3 CONTROL FLOW 18

3.1 BASIC CONTROL STRUCTURES 18

3.1.1 Sequence 18

3.1.2 Exclusive Choice (Branching) 19

3.1.3 Concurrency 20

3.1.4 Synchronisation of concurrent paths of execution 21

3.2 MODELLING SHORTCUTS 28

3.2.1 Merging of Alternative Branches 28

3.2.2 Aggregate Subprocess 31

3.2.3 Iteration 34

4 ADVANCED CONTROL STRUCTURES 41

4.1 ARBITRARY SEQUENCE 41

4.2 ARBITRARY SEQUENCE WITH PARTIAL ORDER 42

4.3 SYNCHRONISATION EXCEPTION 43

4.4 EXCLUSIVE SYNCHRONISATION 45

4.5 RELAXED CONCURRENCY 47

4.6 VARIABLE NUMBER OF CONCURRENT INSTANCES 48

5 LANGUAGE SPECIFICATION 53

5.1 META MODEL 53

5.2 LANGUAGE CONCEPTS 63

5.3 CONCRETE SYNTAX 71

6 EXAMPLES **89**

6.1 BUSINESS PROCESS MAP 89

6.2 PROCESS (DE-) COMPOSITION DIAGRAM 90

6.3 PROCESS INHERITANCE DIAGRAM 90

6.4 ORDER MANAGEMENT 91

6.4.1 Current Process 91

6.4.2 Alternative Version 92

6.5 PROCUREMENT (ECOMOD) 93

6.6 "LIGHT" NOTATION 95

7 EVALUATION OF THE MEMO ORGML **97**

8 CONCLUSIONS AND FUTURE WORK **110**

REFERENCES **111**

Figures

Figure 1: Key modelling concepts and corresponding levels of abstraction 5

Figure 2: Illustration of Associations between Business Process Types (preliminary notation) .. 6

Figure 3: Generic meta taxonomy of events 8

Figure 4: Revised taxonomy of events in the context of business processes 12

Figure 5: Illustration of order for combining event symbols (preliminary)................................... 13

Figure 6: Marking an event as overloaded (preliminary) 14

Figure 7: Symbols to represent starting and terminating events (selection, preliminary) 14

Figure 8: Taxonomy of exceptions 16

Figure 9: Example for assigning an exception type to a process type (preliminary)........................ 17

Figure 10: Illustration of sequence 19

Figure 11: Example representation of two way branching 20

Figure 12: Representation of parallel execution 21

Figure 13: Illustration of synchronisation rules 24

Figure 14: Conjunctional synchronisation of parallel paths.. 25

Figure 15: Conjunctional synchronisation of different levels of parallelisation 25

Figure 16: Synchronisation of synchronised parallel paths 25

Figure 17: Combination of conjunctional and disjunctional synchronisers 26

Figure 18: Irregular combination of conjunctional synchronisers 27

Figure 19: Irregular combination of disjunctional synchronisers.. 28

Figure 20: Merging of two alternative paths, starting with common event 29

Figure 21: Merging of two alternative paths, starting with common process 29

Figure 22: Merging two exclusive choices 30

Figure 23: Partial merging of two alternative exclusive choices... 31

Figure 24: Aggregated process as replacement of sequence 32

Figure 25: Representation of branching processes... 33

Figure 26: Replacement of parallel paths 33

Figure 27: Example of prohibited use of aggregated process 34

Figure 28: Representation of an iteration without specific concepts 34

Figure 29: Representation of nested iterations without specific concepts 35

Figure 30: Example of an iteration with "repeat until" logics 36

Figure 31: Examples of multiple alternative events following an iteration.................................. 36

Figure 32: Example of a "while" control structure 37

FIGURE 33: NESTED ITERATIONS 37

FIGURE 34: PARALLEL PATHS OF EXECUTION WITHIN AN ITERATION BODY 38

FIGURE 35: NESTED ITERATION WITH BRANCHING 39

FIGURE 36: SYNTACTICALLY WRONG REPRESENTATION OF NESTED ITERATION 39

FIGURE 37: CORRECT REPRESENTATION OF NESTED ITERATION 40

FIGURE 38: EXAMPLE OF ARBITRARY SEQUENCE 41

FIGURE 39: EXAMPLE OF PARTIAL ARBITRARY ORDER 42

FIGURE 40: USE OF A SYNCHRONISATION EXCEPTION 44

FIGURE 41: EXAMPLE OF AN EXCLUSIVE SYNCHRONISATION (PRELIMINARY) 46

FIGURE 42: ALTERNATIVE REPRESENTATION WITH ADDITIONAL FINAL EVENT (PRELIMINARY) 46

FIGURE 43: RELAXED CONCURRENCY, CONCURRENCY EXCEPTION AND EXCLUSIVE SYNCHRONISATION (PRELIMINARY) 48

FIGURE 44: EXAMPLE OF MULTIPLE CONCURRENT INSTANCES WITH CONJUNCTIONAL SYNCHRONISATION (PRELIMINARY) 50

FIGURE 45: MULTIPLE CONCURRENT INSTANCES WITH ALTERNATIVE SYNCHRONISERS (PRELIMINARY) 50

FIGURE 46: RESTRICTED CONJUNCTIONAL SYNCHRONISATION (PRELIMINARY) 51

FIGURE 47: DIFFERENCE BETWEEN SPECIFICATION STYLES 53

FIGURE 48: MEMO ORGML META MODEL - FOCUS ON PROCESSES 56

FIGURE 49: SYMBOLS TO REPRESENT ASSOCIATIONS BETWEEN BUSINESS PROCESS TYPES 86

FIGURE 50: COMMENTS AND CONSTRAINTS 88

FIGURE 51: CONNECTOR 88

FIGURE 52: BUSINESS PROCESS MAP 89

FIGURE 53: ILLUSTRATION OF PROCESS (DE-) COMPOSITION DIAGRAM 90

FIGURE 54: ILLUSTRATION OF PROCESS INHERITANCE DIAGRAM 91

FIGURE 55: EXISTING ORDER MANAGEMENT PROCESS 92

FIGURE 56: ALTERNATIVE DESIGN OF ORDER MANAGEMENT PROCESS 92

FIGURE 57: HIGH-LEVEL PROCESS 93

FIGURE 58: MULTI-PERSON EVALUATION OF OFFERS 94

FIGURE 59: MANUAL ORDER ENTRY 95

FIGURE 60: "LIGHT" NOTATION - EXAMPLE DIAGRAM 1 96

FIGURE 61: "LIGHT" NOTATION - EXAMPLE 2 96

Tables

TABLE 1: DIFFERENT TYPES OF SUBPROCESSES (PRELIMINARY NOTATION) 7

TABLE 2: GENERIC CATEGORIES OF EVENTS 9

TABLE 3: SYMBOLS TO REPRESENT BASIC CLASSES OF EVENTS (PRELIMINARY) 13

TABLE 4: SYMBOLS TO REPRESENT BASIC CLASSES OF EXCEPTIONS (PRELIMINARY) 16

TABLE 5: SYMBOLS TO REPRESENT TYPES OF BRANCHING DECISIONS (PRELIMINARY) 19

TABLE 6: SYMBOLS TO REPRESENT THE NUMBER OF EXECUTIONS (PRELIMINARY) 49

TABLE 7: AUXILIARY TYPES 55

TABLE 8: CONSTRAINTS 63

TABLE 9: COMMENTS ON *ANYPROCESS* 65

TABLE 10: COMMENTS ON BUSINESSPROCESS 66

TABLE 11: COMMENTS ON CONTROLFLOWSUBPROCESS 69

TABLE 12: COMMENTS ON EVENT 70

TABLE 13: ELEMENTS OF THE GRAPHICAL NOTATION: PROCESS SYMBOLS 72

TABLE 14: SYMBOLS TO REPRESENT EVENTS (UPPER SYMBOL: "MATT", LOWER SYMBOL: "GLOSSY" VARIANT) 73

TABLE 15: REPRESENTATION OF EXCEPTIONS (UPPER SYMBOL: "MATT", LOWER SYMBOL: "GLOSSY" VARIANT) 73

TABLE 16: REPRESENTATION OF COMPOSED EXCEPTION TYPES 74

TABLE 17: REPRESENTATION OF BRANCHINGS AND MERGERS 76

TABLE 18: REPRESENTATION OF SYNCHRONISERS 77

TABLE 19: REPRESENTATION OF ITERATIONS 79

TABLE 20: REPRESENTATION OF VARIABLE NUMBER OF CONCURRENT INSTANCES 80

TABLE 21: RELAXED CONCURRENCY, CONCURRENCY EXCEPTION AND EXCLUSIVE SYNCHRONISATION 81

TABLE 22: REPRESENTATION OF SYNCHRONISATION EXCEPTION 83

TABLE 23: REPRESENTATION OF ARBITRARY SEQUENCES 84

TABLE 24: PROCESS COMPOSITION/DECOMPOSITION 84

TABLE 25: SYMBOLS FOR REPRESENTING PROCESS SPECIALISATION 85

TABLE 26: ASSIGNING TASKS TO PROCESSES 87

TABLE 27: GENERIC REQUIREMENTS FOR DSML 99

TABLE 28: GENERAL REQUIREMENTS FOR ORGANISATION MODELLING 101

TABLE 29: SPECIFIC REQUIREMENTS 109

Typographical Conventions

If textual elements of meta models (or the meta meta model respectively) are referred to in the standard body text, they are printed in Arial, e.g. OrganisationalUnit.

Acknowledgements

Sebastian Bittmann provided valuable support with the specification of constraints that supplement the meta model. Nevertheless, I am a responsible for any possible errors.

1 Introduction

Models of business processes are widely accepted as a key abstraction of any organisation. This is for various reasons. Firstly, business process models are literally a representation of how an organisation works on the operational level. Hence, they serve a documentation function for those, e.g. new employees, external consultants etc., who are interested in understanding essential tasks in an organisation and how they are performed. This does not only relate to the flow of control that characterises a process, but also to the resources it requires, the risks it holds and the goals, it is aimed at. Secondly, business process models are both subject and instrument of analysing and eventually improving the efficiency of an organisation. They support to focus on those aspects that are relevant for efficiency, e.g. provision of resources, execution time, bottlenecks, etc. Thirdly, they provide a conceptual foundation for the design of software systems that support executing and monitoring business processes. They also support the collaboration of business people and IT experts since they provide a representation of an organisation's action system that is regarded as comprehensible by most stakeholders. Hence they serve as a common reference, thereby supporting to overcome frequent language chasms. Finally, business process models contribute to a more focussed and customer-oriented perception of an organisation's action system. A business process is directed towards an outcome that is supposed to satisfy an external or internal customer. Different from traditional functional division of labour, where – in the extreme case – every employee saw only the function he was in charge of, a business process constitutes a coherent pattern of work that does not only direct the various functions towards a common goal, but that can also influence the employees attitude to become more goal- or customer-oriented.

Against this background, it is not surprising that business process modelling in general, business process modelling languages in particular have gained remarkable attention both in research and practice. There is a plethora of approaches that originate in computer science and that focus on a sound formal foundation for business processes. On the one hand, they are aimed at proving certain features of a process, e.g. that it does not include any deadlocks. On the other hand, they intend to provide executable models. The most notable example for approaches of this kind are Petri nets, which come in many different flavours, some especially designed for workflow or business process modelling (e.g. Van der Aalst 2002). While formal languages provide clear advantages with respect to the consistent execution of business processes and hence to the integrity of a corresponding information system, they usually lack an elaborate representation of business-related aspects, such as the differentiation of specific types of processes or resources. Event-Driven Process Chains (EPC) put more emphasis on the support of business-oriented analysis. There were a few attempts to formalise them (e.g. Wehler and Langner 1998, Rittgen 2000) as well as principal obligations concerning the feasibility of such attempts (Van der Aalst et al. 2002). The language has evolved mainly through extensions of a corresponding toolset. It still includes concepts that lack a specification of their semantics, e.g. resource "types" that are reduced to the provision of graphical symbols. As a consequence of the increasing relevance of workflow management systems, a few initiatives emerged that target the standardisation of workflow

specifications. The *Business Process Execution Language for Web Services* (BPEL4WS, Andrews et al. 2003) is an application of XML schema that is propagated by a coalition of large IT vendors. It is a successor of the *Workflow Process Definition Language* (WPDL, Coalition 1996), which was proposed by the Workflow Management Coalition. It serves to specify schemas of workflow management systems. It does not include a graphical notation. A further initiative under the umbrella of the OMG was created to fill that gap: The *Business Process Modelling Notation* (BPMN) started to supplement a graphical notation for representing business processes, which are specified by a specification language as BPEL4WS (OMG 2008). In a later version BPMN includes a specification of the execution semantics itself (OMG 2009). As a consequence, it does not depend on BPEL anymore for specifying executable workflow models. While BPMN includes a plethora of concepts that allow accounting for peculiarities of business processes, the specification remains dissatisfactory. On the one hand, it often lacks a precise terminology, e.g. a clear differentiation of levels of abstraction. On the other hand, the graphical notation itself hardly fulfils its promise to provide business people with an intuitive representation of business processes: The symbols remain abstract, without any visualisation of business-related aspects. Börger and Thalheim 2008) present a "framework" for the specification of business process modelling languages. It consists of a formal, rule-based meta language and its application to generic control structures of business processes. In a case study, the authors demonstrate its application to BPMN. While this makes it more convenient to use than mere formal languages such as Petri nets or process algebras, its emphasis is on the specification of the execution model, too – abstracting from specific business-related characteristics.

None of the existing approaches fits exactly the purpose a process modelling language should serve within the MEMO framework. Except for EPCs they are not specified through meta models, which would make it cumbersome to integrate them with the MEMO modelling languages. At the same time, EPCs are not a language to be conveniently integrated either, since there seems to be no comprehensive specification. A few excerpts of a corresponding meta model can be found in Scheer 2001). However, they seem to serve mainly illustration purposes. At the same time, the specification the ARIS toolset is based on is not available. Within MEMO, process modelling is part of the Organisation Modelling Language (OrgML). It is primarily aimed at providing elaborate support for analysis and design of action systems and corresponding information systems. Therefore, it emphasises the integration with relevant parts of an enterprise model such as required IT artefacts, business goals, organisational units etc. The specification is not aimed at specifying corresponding execution semantics. However, it should be sufficient for mapping respective business process models to representations that are executable.

The specification of a domain-specific modelling language (DSML) is a demanding task. A DSML can be an artefact of remarkable complexity. Often, it will be even more demanding that requirements are difficult to analyse because the prospective users have not clear idea of what to expect from a DSML. This is different with business process modelling, since there is remarkable experience with developing and using business process models. Nevertheless, the design of a business process modelling language requires accounting for requirements. A corresponding requirements analysis has been presented in a previous report (Frank 2011b) that covered the scope of the entire OrgML, i.e. modelling of

organisation structure and processes. For many, reading the specification of a DSML is not exciting – to say the least. Therefore, the first part of the report is focussed on descriptions of language concepts that are illustrated with examples. It is divided in basic concepts and control flow concepts. The second part comprises the meta model. It can be omitted by readers who are more interested in using the language than in building tools. The graphical notation is introduced on the fly with the illustration of the language concepts. An addition to that, a comprehensive representation of all notation elements is provided together with a selection of example models. The language specification presented in this report does not cover the information flow within a business process. Information flow or information logistics is a field of remarkable complexity, if one wants to allow for elaborate models that. Therefore, the specification will be extended with corresponding concepts in a separate report.

This report is not intended to serve as a mere handbook for guiding the use of the OrgML. Instead, it documents the design of the language including the discussion of related challenges and design problems. Therefore, a preliminary graphical notation is used before the language itself is specified. Even though it is in part very similar to the final notation, it should not be mistaken for the that. It serves to illustrate concepts and related design issues as a foundation for the subsequent language specification. Those readers who are interested in using the language only may want to skip chapters 2 to 4 and focus on the remaining chapters. Also, the meta model is required only for those readers who want to get a deeper understanding of the language specification.

2 Basic Concepts

The main purposes, the MEMO OrgML should serve with respect to modelling business processes are outlined in the requirements SR9 to SR28 in (Frank 2011b). Different from most approaches that are aimed at a sound formal specification of process modelling languages, the MEMO OrgML is not primarily focussed on the specification of process execution models. Instead, the emphasis is on various types of analysis and design. Since MEMO business process models are also intended to serve as a conceptual foundation for building process-oriented software, such as WFMS, they are supposed to incorporate a sufficiently precise specification of their semantics in order to allow for transformations into languages that are used for implementation purposes, e.g. workflow specification languages. An example for such a transformation based on a previous version of the MEMO OrgML is presented in Jung 2004). By default the language concepts serve to specify types, e.g. event types, subprocess types etc. Note that we will often speak of events, subprocesses etc. without adding an explicit "type".

To develop a business process modelling language, a definition of the term business process as the one presented in Frank 2011c, p. 5) may serve as a starting point. However, it is not sufficient. A business process is composed of processes. To differentiate these processes from the entire business process, we call them *subprocesses*. An elementary subprocess is an elementary unit of work within a business process, i.e. it is not supposed to be further decomposed. Subprocesses can be composed to aggregate subprocesses. Subprocesses are arranged with respect to a certain flow of control that defines the sequence of executing processes. A process can be described by its outcome, the resources it requires, the actors that are responsible for it and the tasks it involves. To compose a set of subprocesses to a business process, it has to be specified how their execution is synchronised. This requires a conception of time – to express *when* a process will be started and what happens when it terminates. This conception can be realised through the notion of an *event* and relationships between events that express their relative position on a common time axis. Furthermore, events can serve the purpose to notify agents (systems or humans) that have subscribed to be informed about certain aspects of a business process' states or state changes. *Exceptions* are a special form of events. Different from events they are not expected to occur on a regular base. They could be modelled as regular events. However, that would not only increase the complexity of a model, it would also corrupt its comprehensibility by distracting the observer from the regular pattern of execution.

The analysis of concepts to model organisational structures showed that most of them cannot be directly represented as meta types (Frank 2011c). This is clearly different with the key concepts used for process modelling. Figure 1 illustrates their use on various levels of abstraction. Apparently, the core meta types "Process", "Event" and "Exception" clearly fulfil the criteria referred to in modelling Rule R2 (Frank 2011d).

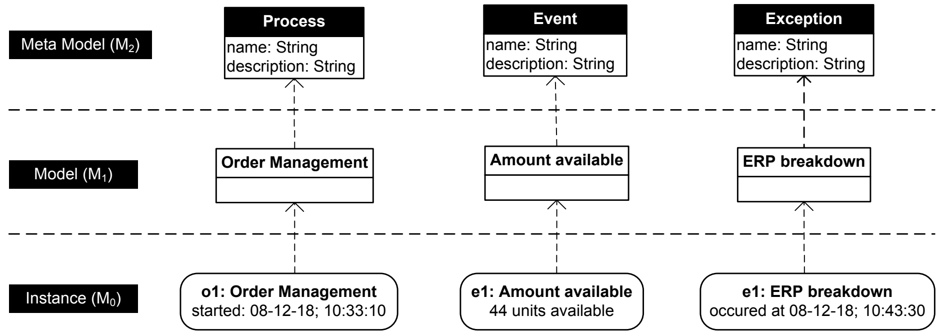


Figure 1: Key modelling concepts and corresponding levels of abstraction

To allow for a more elaborate analysis and construction of process models, more specific language concepts are required – in other words: The vocabulary that is provided with the modelling language needs to be extended. For this purpose, the key concepts "Subprocess", "Event" and "Exception" are differentiated into more specific concepts. This conceptual differentiation may result in further meta types, i.e. in technical terms represented within the OrgML, or it may produce a classification schema which serves to differentiate relevant shapes of the core terms. The following discussion of design issues makes use of graphical notations for illustration purposes. Note that the respective symbol are only in part the same as those that will finally be presented as the graphical notation of the language.

2.1 Process

The generic term "process" is at the core of the targeted DSML. It comprises entire business processes, subprocesses and aggregate processes.

2.1.1 Business Process

Usually business process modelling is focussed on one business process type at a time only. While this makes sense with respect to reducing complexity (and separation of concerns), it is not satisfactory, if there are interdependencies between process types. These interdependencies concern resources, the instances of various process types use (or compete for), events, exceptions, etc. Accounting for these interrelationships fosters the overall integrity of the corresponding organisational system (e.g. integrity of business rules, positions, roles etc.) and the information system (e.g. commonly used classes, events, exceptions, exception handling etc.). Therefore, the OrgML includes the concept "Business Process" and a corresponding symbol. A business process type is comprised of subprocesses, events and a corresponding control flow. In addition to its internal structure, a business process type can be characterised by associations to other business process types. Currently, they comprise two directed and two undirected associations. A business process type may support another one or it may be a special case of another one. Two business process types can be similar or they may compete. Note that there is no special semantics of these association types defined. They only serve to document relationships between business processes in order to foster transparency and integrity. If a business process type is defined to be a special case of another business process type, there will usually be no need to

express that they are similar. However, there is no constraint that enforces the mutual exclusion of these association types, since they may be based on different kinds of commonalities. The business process map in Figure 2 illustrates the use of these associations.

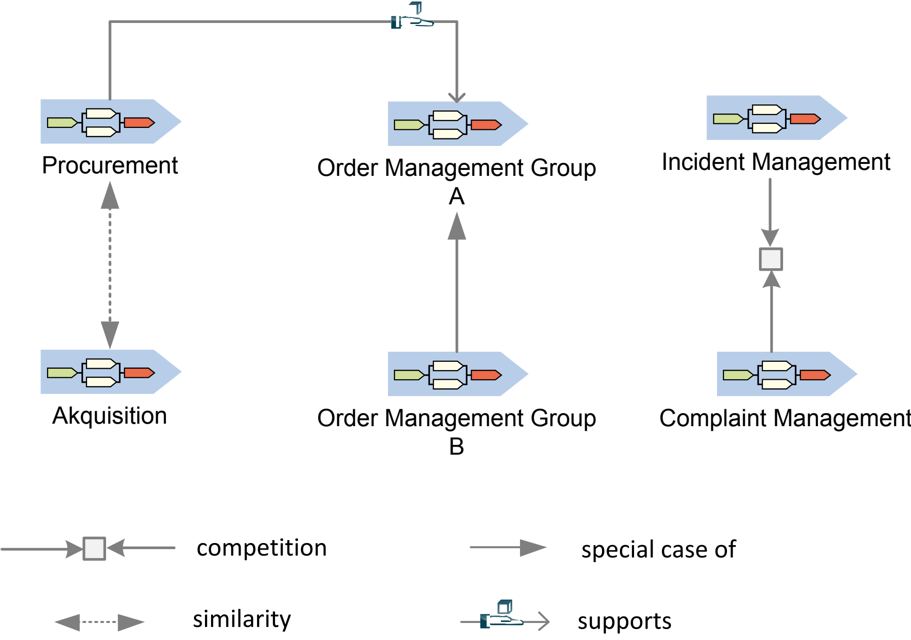


Figure 2: Illustration of Associations between Business Process Types (preliminary notation)

2.1.2 Subprocess

Subprocesses can be differentiated with respect to the level of automation: "Manual Process", "Computer-Supported Process" and "Automated Process". Note that the notion of an automated process implies that it is triggered automatically, too – while a computer-supported process might be triggered automatically or manually. The concept "Any Process" is used for those cases where one does not know or does not want to care about the specific characteristics of a process. Furthermore, there is the concept "External Process". An external process is executed under the control and responsibility of somebody outside of the organisation – which does not exclude that it takes place physically in the organisation's facilities. It may be an entire business process within the external organisation. However, from the perspective of the local business process it is a subprocess. Since the two dimensions "level of automation" and "internal/external" are orthogonal, they can be combined to eight different kinds of subprocesses. However, in most cases, it will not be advisable to account for the peculiarities of an external process. Instead, one would usually take advantage of the abstraction that results from encapsulating external processes. Table 1 shows the concepts together with the symbols used to visualise them. There is no doubt that all these concepts satisfy the criteria proposed with **Modelling Rule R2** (Frank 2011d). Hence, they are suited as concepts of a modelling language. Note, however, that does not necessarily mean to specify them as meta types: Firstly, a resulting concept may not correspond to any term of the relevant technical terminology. Secondly, the resulting number of terms may overburden prospective users. Thirdly, this "type" of a process may be changed during the life time of

a model. That could recommend using pseudo types which are differentiated by entity states (**Modelling Rule R3, Frank 2011d**). Also, it seems reasonable to assume that there are further specific concepts which are used in certain domains, e.g. in the financial services sector or in health care. Hence, it is important to provide for corresponding extension mechanisms.

Any Process	represents process types which are not specified any further	
Manual Process	represents process types that are executed by humans without the support of computers	
Computer-Supported Process	represents process types that are executed by humans that use computers	
Automated Process	represents process types that are automated	
Any External Process	represents process types external agents are responsible for; note that this may be an entire business process type within the organisation of the external agent.	

Table 1: Different Types of Subprocesses (preliminary notation)

2.2 Event

The idea of an event is essential for dynamic abstractions. Events are are used to couple processes to a more complex process: A process is triggered by an event and it produces or enables one or more further events, which in turn may trigger further processes. Similar to "process", the term "event" is frequently used both in everyday's language and in many technical terminologies. Therefore it is not surprising that its meaning is vastly overloaded. At the same time, "event" is regarded as a key term of philosophical ontology. However, philosophy has not produced a uniform conception of event either. There is a plethora of considerations from different viewpoints, based on different ontological traditions. Therefore, the ontological notions of event are characterised by a remarkable, sometimes subtle diversity. They include questions such as "does an event have a temporal duration?", "what is the difference between state and event?", "is there a categorial difference between objects and events?" etc.

In the context of business process modelling we do not need a notion of event that satisfies ontological requirements, since our modelling concepts are not intended to describe the world as it is, but to foster abstractions of business processes that satisfy certain purposes. Nevertheless, the ambiguity of the term recommends to thoroughly define a conception that is suitable for modelling business processes with respect to relevant purposes. There is a multitude of different event types that may occur in business processes. The question is, which ones are suited as concepts to be included in the modelling language. To develop an answer, we will first look at a comprehensive – however, not complete –

taxonomy of events in the context of business process modelling. Subsequently, we need to decide which of these concepts should be included in the modelling language.

2.2.1 A Generic Taxonomy of Events

In general, an event within a system – action system and/or information system – can be regarded as a change of state. There are three generic kinds of change: A set of new objects can be created, the state of a set of objects changes, or a set of objects is deleted. State may relate to the state of a particular object, e.g. the state of the attribute "revenues" of an object of the class "Product". It may also relate to an aggretated value, e.g. the sum of all corresponding revenues. Furthermore, it may reflect a relation between different states, e.g. "revenue A > revenue B", or a function, e.g. "margin" which is calculated from revenues and costs.

This preliminary conception of event would include information changes such as the modified content of a database or the creation of a new file. It would also comprise physical objects such as parts or products that were built, modified or disposed. From a materialist point of view, it would even include decisions – which would be regarded as the state changes of the human brains that are involved or that are affected. Furthermore, events can be generated through time, either points in time or time periods. This seems to be not compliant with the definition suggested above: The progress of time is independent of a system's state – the implications of the theory of relativity are certainly not relevant for our purpose. However, time can be regarded as a dimension that characterises a system: A system at time t_0 is different from the same system at time t_1 – even it its state has not changed. A good example for this assertion is the feature "age". While the state of an object may not have changed at first sight, the function "age" will deliver different values with progressing time.

This generic conception of event results in the meta taxonomy illustrated in Figure 3. Note that "physical resource" comprises both physical objects such as a computer that was repaired (e.g. within an incident management process) and humans (e.g. the enlistment of a new employee within a process operated by a Human Resources department). "Action" on the other hand refers to human actions that express change within the related human brains, e.g. the result of a decision, the approval of a contract etc. An action may induce a change of corresponding information objects or physical resources, but it does not have to.

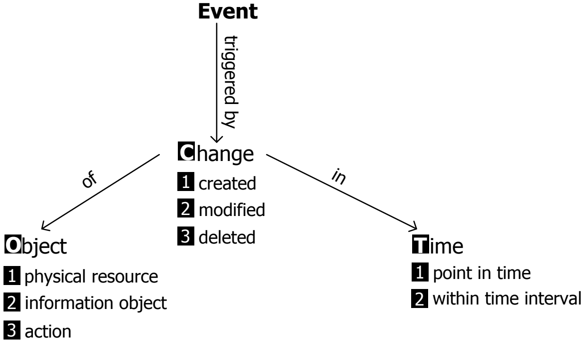


Figure 3: Generic meta taxonomy of events

Since the three dimensions "object", "change" and "time" are (largely) orthogonal, they could be combined to 3x3x2=12 different generic kinds of events. Table 2 illustrated these resulting kinds of events by examples. Note that the dimension "time" is used in a few examples only.

		Physical Resource		Information Resource		Human Brain/Decision/Action
new		A new PC is installed.		An invoice is created.		A customer calls to give an order.
				An order is created.		A customer calls to ask a question that concerns a product.
		A new employee is enlisted.	T1	A christmas card for a customer is created before Dec. 14th.	T1	A department manager signs a christmas card to a customer before Dec. 14th.
				New products are entered into the database.	T1	A customer buys a product (new contract) before the special price expires.
modified	T1	A PC hard drive is replaced within the warranty period.		An order is modified.		A customer calls to modify a previous order.
		An employee is qualified for an ERP system.		Prices of certain products are changed.		A sales assistant confirms a modified order.
						A department confirms a special price for a certain customer.
		A PC memory is extended.	T2	The annual revenues of a customer have exceeded a certain value.		A department denies a special price for a certain customer.
deleted		A PC is removed.	T2	An order is cancelled in time.		A customer sends a fax to cancel a previous order.
				An invoice is cancelled.		
	T2	An employee is laid off within the probation period.		Completed orders are removed from the database.		A customer calls and points out to a mistake in an invoice.

Table 2: Generic categories of events

Analysing the taxonomy of events illustrated in Table 2 leads to the following insights.

- A materialist view is not appropriate. While it may be regarded as a convincing intellectual construction, it does not correspond with conceptualisations prospective users are familiar with (requirement U1). Hence, instead of using the conception of changing brain states, the language should offer concepts that represent the results of human actions more intuitively.

- For executing business processes, information is essential. On the one hand, business processes require information, on the other hand, they produce information. Therefore, analysing how information is handled within a business process is a pivotal task. This does not only include information states and corresponding events, but also other aspects such as the representation and delivery of information as well as awareness. Therefore, a further differentiation of events that are related to these aspects is recommended.
- Those events that are produced by human action (or decisions) often correspond to changes of information they target. Therefore, accounting for both kinds of events, e.g. the confirmation of an order by a human agent, and the resulting change of an order would result in redundancy or in ambiguous guidelines for using the corresponding concepts.
- While many human actions/decisions within a business process will result in information (changes) that represent them, there are actions that may cause relevant events but that are not necessarily documented. If, for instance, a customer calls for requesting information about specific technical features of a product, this will not always be documented. However, when such an event triggers a process, it is assumed that the request is somehow represented in the information system – be it digital or non-digital.
- The manipulation of physical objects is usually not a subject of business processes. Therefore, there is no need to offer specific concepts of events that correspond to certain changes of physical objects. If handling physical objects is an issue in a business process, it seems reasonable to assume that there is a corresponding information system that represents the relevant states of the objects. Also, with the growing dissemination of micro transponders, such as RFID chips, the difference between physical objects and the information objects that represent them gets more and more blurred. Nevertheless, extending the language for other domains, e.g. for modelling manufacturing processes, may recommend to include conceptions of events that represent results of specific operations on physical resources.
- Often, temporal aspects will not be relevant for an event to occur. Therefore, it should not be required to always account for them.

To summarise, the analysis of our first draft of a taxonomy of events suggests to widely fade out some aspects, such as human actions/decisions (not the effects of these) or the manipulation of phyisical objects. At the same time, the outstanding relevance of information handling recommends more differentiated conceptions of events that are related to changes of information state. These additional aspects should be relevant for performing the process.

2.2.2 Conceptualiation of Events

The main purpose of an event is to trigger the appropriate subsequent process. For analysing how well an event is suited to fulfil this purpose, notification is essential. With respect to manual or semi-automated processes that are not triggered automatically, this includes the question how the human agent who is in charge of the subsequent process is notified. With respect to automated processes, it may be important to know whether an event gets propagated to the software or whether the software needs to check for relevant state changes. The refined taxonomy in Figure 4 shows a corresponding classification of events. Since change is mainly related to information objects, there is no further differentiation of the objects of change. In case a human actor is in charge of the subsequent process, the notification about an event can be classified into synchronous (e.g. telephone call, face to face) or

asynchronous (e.g. fax, e-mail, mail). It does not make much sense to differentiate between face to face and telephone calls. Both are synchronous notifications and usually it should be no problem to use the one that is most appropriate. With respect to asychronous communication, it makes sense to differentiate between electronic and traditional media: An incoming electronic message is clearly easier to detect by a machine than a physical message. In the case of an incoming fax, it depends on the technology used to receive the fax. In case of a fax machine, the message is delivered on traditional media, i.e. paper. If a fax server is used, the fax can be regarded as an electronic message. Note that distinguishing synchronous from asynchronous communication is sometimes not trivial. If, for instance, a user is notified by a message in a pop-up window, this could be a synchronous message, if the user perceives the message as soon as it is delivered. It could also result in a asynchronous message, if the user realises the event only later. One could regard synchronous as a special case of asynchronous communication, where the time for storing a message is zero. With a growing amount of time a message is stored it gets more and more asynchronous. Having this in mind suggests to make use of additional comments at least in cases where a message may be on the borderline between synchronous and asynchronous.

With respect to software, an event can be published into the space (e.g. a queue) of an event management system with a corresponding notification of those software components that subscribed for this kind of event. An alternative option would be that an event is not published, but is instead detected by polling the states of the related objects. For instance: A software system could check from time to time whether a particular file or a certain database table had been changed. In both cases, the software that receives a published event or detects an event by some polling procedure will be able to trigger the subsequent process(es). Note that the notification of software and of human agents do not exclude each other: It is possible that an event is propagated to some control software and that a human being is informed about the event, too. If information state changed, one may not want to bother with a further differentiation. Hence, "not specified" is to indicate that some change happened. This generic form applies to the notification of software or human agents, too: Even if one does not want or is not able to specify a certain kind of notification, it may still be relevant to express that there is some kind of notification. This is different with the dimension "Time": If the event is neither related to a point in time, nor to a time interval, it does not make sense to assign a symbol that would express something like "time matters".

According to the taxonomy in Figure 4, events can be classified with respect to the generic type of change of a related information object, to the notification mechanism that is used to communicate them and to related temporal aspects. While all these different aspects can be combined into a particular classification, they can also be used alone, if other aspects are not specified because they are not relevant or no related information is available.

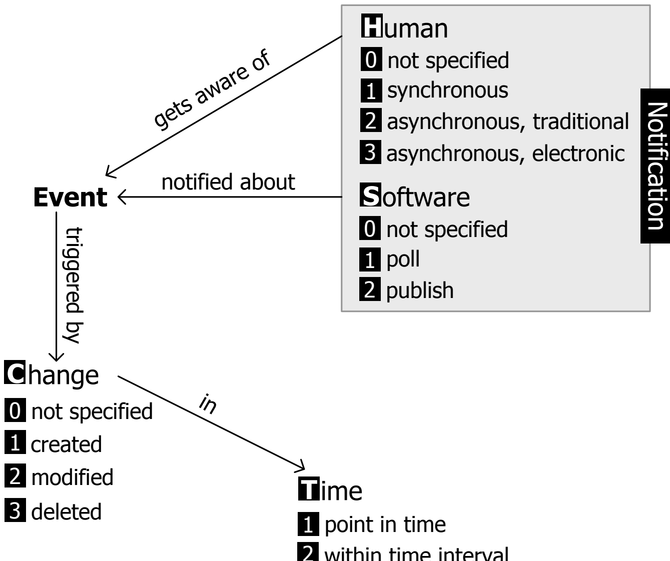


Figure 4: Revised taxonomy of events in the context of business processes

Applying the taxonomy in Figure 4 results in 4x4x3x2 = 96 different classes of events. While differentiating these classes would promote the analysis of business processes, it is probably not appropriate to represent each of them as a meta type. The resulting plethora of terms would certainly not correspond to the technical language prospective users are familiar with – and would probably overburden most of them. Therefore, the language users are provided with the classification criteria only. It is up to them to what degree they want to make use of these criteria (requirement U3). The graphical notation of the language supports this approach by allowing for combining the corresponding symbols. Table 3 illustrates the construction of notation elements that serve to render event types. Sometimes, it will not be clear whether an information change is to be characterised as the creation of a new object or as the modification of an existing object. Take, for example, the event that occurs when the revenues of a customer exceed a certain value. While this event corresponds to the modification of an object, it also corresponds to the creation of an invoice object. Therefore, in cases of ambiguity it is recommended to make use of additional comments.

Table 3 shows the different shapes of the basic event types.

Change	not specified	new	modified	deleted
Time	not specified	point in time	time interval	

Notification: Human	not specified	synchronous	asynchronous, traditional	asynchronous, electronic
Notification: Software	not specified	poll	publish	

Table 3: Symbols to represent basic classes of events (preliminary)

To allow for a more differentiated graphical representations of events, these symbols can be combined according to the taxonomy shown above. To avoid a plethora of redundant symbols, the following rule applies to the construction of graphical symbols that represent event types. If an event type is characterised by multiple dimensions, the primary symbol is selected in the following order: information change, time, notification. The examples in Figure 5 illustrate this rule.

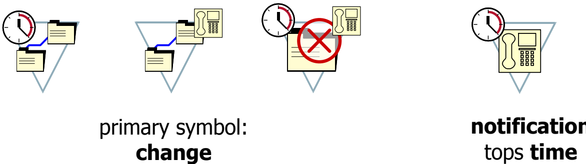


Figure 5: Illustration of order for combining event symbols (preliminary)

Note that the classification shown in Table 3 covers two different aspects of an event: the act of change that constitutes the event in the first place and its notification, which may generate a further event at a later time. It is left to the modeller whether to do without any classification, to use only one aspect or to combine both for classifying an event (as shown in Figure 5). However, a modeller should be aware of the difference in order to avoid confusion. The event that is related to a notification mechanism is meant to occur when the notification is received, not when it is sent. Sometimes, distinguishing these two events will not be significant: if the mode of receiving a notification is implied by the mode of sending it. If the mode of sending a notification allows for more than one mode of receiving it, it is important to be aware of the above determination. With respect to current technology, the most obvious example for such as case is sending a fax, which can be received either by a traditional fax machine or a fax server. It may be regarded as important to distinguish these two modes of reception since they have different implications on controlling a workflow and on information logistics. One event can be transmitted by making use of more than one notification mechanism: If an event triggers a concurrent split, it may be that the following concurrent processes are notified differently. For instance: A process that results in a new state of some document produces an e-mail message to communicate this event to some of the subsequent concurrent processes and a fax to communicate it

to other subsequent processes. However, only one event symbol can be used that applies to all subsequent concurrent processes. Therefore, it is possible to use the event symbol that seems most appropriate (in the above example this could be a symbol for electronic fax) and tag it as "overloaded", which means that other notification mechanisms are used as well (see Figure 6).

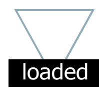


Figure 6: Marking an event as overloaded (preliminary)

Events that trigger a business process as well as those that terminate one are represented by special symbols that can be combined with any other event symbol. As with intermediate events, it is possible to use the starting and terminating symbols in a generic way without any further specification. Figure 7 shows a few examples of starting and terminating events.

Figure 7: Symbols to represent starting and terminating events (selection, preliminary)

The classification of events suggested above is still rather generic and does not account for more specific but still common events in business processes such as, e.g., "signed", "confirmed", "denied". Currently, these types of events are not accounted for in the language. This is for two reasons: On the one hand, it is assumed that events of this kind are not of remarkable relevance with respect to analysing business processes (an assumption which may turn out to be inappropriate). On the other hand, it would not be possible to build a complete list of more specific events. Therefore, more specific events are not represented by corresponding language concepts. It is, however, possible to account for them nevertheless: If one defines conventions for naming event types, it is possible to use names as pseudo types.

One might consider excluding that only certain types of evens may trigger cetain process types. For instance: Should it be possible for an event that can be detected by software only (e.g. through polling) to start a manual process? While this may indicate a problem with a process model, we decided to not add a respective constraint. Nevertheless, a tool that supports model analysis may point at such a constellation.

2.3 Exception

An exception is an event that may occur during the execution of a business process. Different from regular events that serve to trigger processes or to terminate a business process, an exception – as suggested by the name – is not generated on a regular base. It is introduced for two reasons: Firstly, it helps to reduce the complexity of business process models by leaving out cases that occur only very rarely. In other words, it puts emphasis on how to accomplish the task without regarding possible, but rare cases of disturbance. Secondly, the concept of an exception provides an abstraction that fosters

reuse and integrity. It demands for categorising/classifying exceptions and for defining specific exception handling processes for each kind of exception. Hence, an exception handling process can be reused in all processes where a certain kind of exception occurs. This does not only contribute to an organisation's consistency as it perceived by users, it also fosters its integrity with respect to changes: If a exception handling process needs to be modified, this happens only once. Of course, these advantages of exceptions are well known from software engineering.

Note that an exception is a modelling concept. Hence, only those exceptions that are anticipated as such are accounted for. Also, there may be conceivable exceptions such as natural disasters that are not represented in a model because the probability of their occurrence is regarded as too low for bothering with them. To support a differentiated representation of exceptions, the OrgML should offer meta types or classifications of exceptions that correspond to useful analysis or design objectives. One important criterion for classifying exceptions is their effect: An exception is *fatal*, if it cause a process to fail. In this case, exception handling is focused on terminating the process and rolling back the corresponding information system. If an exception is a manageable disturbance, exception handling is aimed at "repairing" and resuming the process. Orthogonol to the ultimate consequences of an exception are its causes. There seem to be three different direct causes: technical failure, missing resources or human action. Technical failure includes problems with software and hardware being used in a process as well as breakdowns of power supplies or communication infrastructure. Missing resources relate to missing human operators, missing machinery (e.g. required devices) or missing space. There are multiple patterns of human action (or defaulted human action) that may generate a process exception. To give a few examples:

- An actor decides to abandon the process.
- It turns out that an actor had provided false data at an earlier stage of a process.
- An actor decides to cancel the process.

The classification that is suggested for the MEMO OrgML does not account for specific kinds of actions or the related motives – because a plethora of those are conceivable. Instead, it allows for classifying an exception as being caused by human action or request. Sometimes, it is not possible to decide whether human action or a technical problem caused an exception – e.g. if an exception is assigned to an external process – or there may be both a technical and a human cause and one does not want to bother with treating them separately. For these cases, it is possible to make use of a generic concept of exception. For handling exceptions appropriately, it is mandatory to detect them in time. Therefore, analysing a business process should focus on the question how likely it is to detect a possible exception: If there is a chance for an exception not to be detected in time, this should be a reason to think about appropriate measures to foster the detection of the exception. Figure 8 shows a corresponding taxonomy of exceptions.

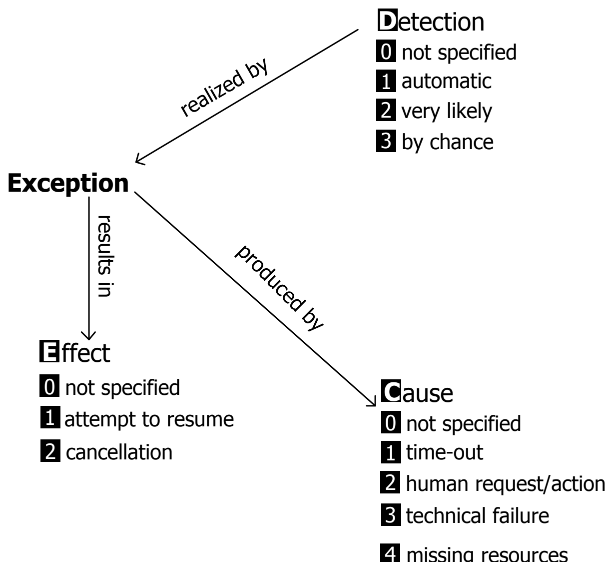


Figure 8: Taxonomy of exceptions

To reduce the amount of different symbols, we faded out "missing resources" as possible cause. If required, in can be represented by "time-out" or "human action" – together with a self-descriptive designation. The graphical symbols proposed to illustrate the resulting classes of exception are shown in Table 4. Note that there are no special symbols for indicating that *cause* or *detection* are not specified.

Effect	not specified	resume	cancel	
Cause	not specified	time out	human action	technical failure
Detection	not specified	automatic	very likely	by chance

Table 4: Symbols to represent basic classes of exceptions (preliminary)

An exception type is assigned to a subprocess type. Figure 9 illustrates the representation of an exception type within a business process model. In case a process is assigned more than one exception, the corresponding symbols should be stacked.

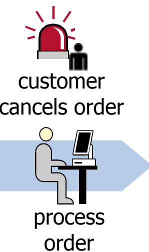


Figure 9: Example for assigning an exception type to a process type (preliminary)

While (sub-) processes and events are essential to define how a business process is executed, they are not sufficient. There is need for further concepts that allow for specifying the temporal (or causal) order that determines the execution of a process. These concepts are usually referred to as *control structures*.

3 Control Flow

There are only a few basic control structures that are part of any process modelling language. The term *process* can be used either for representing an entire business process or a sub process only. To reduce this ambiguity, we use the term *business process* for the entire business process and the term *process* for any process that is part of a business process. The specification of the corresponding language concepts can be done only after all control structures have been presented.

A business process model can be regarded as a directed graph consisting of certain types of nodes (or vertices). In the simplest case, there were only two types of nodes, process and event, which would form a bipartite graph, i.e. a node of a certain type would never be connected to a further node of the same type. Prominent examples of bipartite graphs are basic classes of Petri nets. While a modelling language should be kept simple, the MEMO OrgML makes use of more than two types of nodes. This is for two interrelated reasons. Firstly, additional types of nodes allow for building structures that are more comprehensible for human observers. Secondly, they reduce the effort required for analysing a business process model, e.g. to check whether a synchronisation of parallel paths of execution is syntactically correct.

Before we start with the description of basic control structures, we will first define a few constraints and features of OrgML business process models.

- A business process model starts with exactly one start event out of possibly many mutually exclusive start events. It is not possible to define two concurrent start events. Justification: If there were more than one concurrent start events, it would leave the business process model with substantial ambiguity. After one of the start events had fired, it would not be defined whether the remaining – concurrent – start events would still have to (or might) fire.
- A business process model is terminated by one or more mutually exclusive stop events. Justification: If it was possible to terminate a business process with two or more concurrent events, there would not be no clear indication, when a process actually terminates. In other words: A terminating event would not necessarily terminate a process.

3.1 Basic Control Structures

There are a few basic concepts to define the control flow of a business process: sequence, branching (alternative execution), parallel execution and synchronisation.

3.1.1 Sequence

A sequence represents the consecutive execution of a set of processes, which are interlinked by events – see example in Figure 10.

Abstract syntax: A sequence consists of event types and process types. Each event type produces either none (in case of a terminal event) or one process type. Each process type produces exactly one event type.

Semantics: Only after a process was terminated, the subsequent event will be generated, which in turn is the prerequisite for starting the next process. As far as synchronisation is concerned, it is not necessary that the termination of a process and the occurrence of the event it produces happen at the same time. For instance: A process may be terminated, but the subsequent event will be generated only after 10 more minutes. Accordingly, an event does not have to occur at the same time a subsequent process starts.

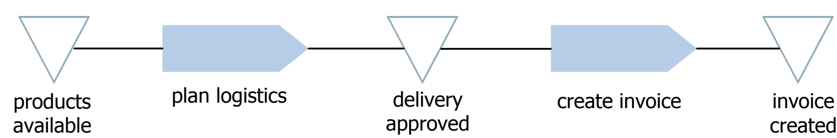
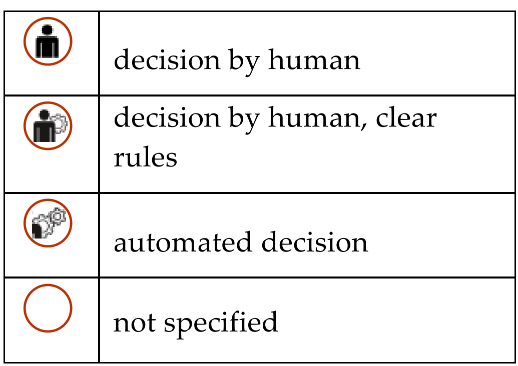


Figure 10: Illustration of sequence

3.1.2 Exclusive Choice (Branching)

An exclusive choice indicates that a business process is split in to or more alternative branches of execution. The actual path of execution is selected during run time by evaluating a condition that is used to control the branching. Note that only one of the possible branches can be selected (exclusive OR). The exclusive choice is represented by a set of mutually alternative events. While the actual meaning of an event cannot be checked formally, it is assumed that the events that form an exclusive choice cover all possible (relevant) events at this point. If, e.g., an exclusive choice produces two events E_1 and E_2, then E_1 equals non E_2. In addition to events, an explicit decision can be added. This can be either a comment or a (formal) expression that can be evaluated to either true or false (in the case of an exclusive choice with two alternatives). The language used to define such an expression will be introduced only later, because it needs to refer to the context of a business process, e.g. to resources, information etc. For analysing the economics of a business process it is important to know, whether the decision the branching is based on is done automatically or by a human. In the latter case, it can be differentiated between decisions that are based on fairly clear rules and those that are more dependent on human expertise. The corresponding symbols are depicted in Table 5.



	decision by human
	decision by human, clear rules
	automated decision
	not specified

Table 5: Symbols to represent types of branching decisions (preliminary)

Note that especially in case of an automated decision it should be possible to represent a formal expression that can be evaluated to calculate the decision. For analysing a business process type – e.g. with respect to the required resources or to costs or time consumption – it may be useful to account for the relative frequency – or probability – that is characteristic for the execution of each alternative produced by an exclusive choice. If corresponding numbers are available, they can be assigned to each alternative (see example in Figure 11).

Abstract syntax: An exclusive choice split is associated with a process that produces it. Each of the n alternative paths that constitute the possible alternatives starts with an event which is associated to the exclusive choice split. With respect to its syntax, each alternative path is treated like an independent business process, i.e. it may contain any control structure that is defined for a business process, e.g. further exclusive choices or parallel paths of execution. An exclusive choice can be assigned an expression that serves to characterise the decision. Each alternative path can be assigned a relative frequency

Semantics: During the execution of a business process, one and only one of the alternative paths produced by a branching can be executed (XOR). The branching condition should be specified in a way that there will be always one alternative that fits the decision criteria. Otherwise it would be possible for processes to get stuck in a deadlock. Note that this semantic constraint cannot always be formalised – and hence not enforced by a tool – because it may require an evaluation of the relevant action system. In the current version of the language, the expression that can be assigned to an exclusive choice is regarded as a comment that serves to describe the underlying decision. In the case of a two way branching it could be interpreted as a Boolean expression. The sum of the relative frequencies – or probabilities – that are assigned to the alternative branches of an exclusive choice must be 1.

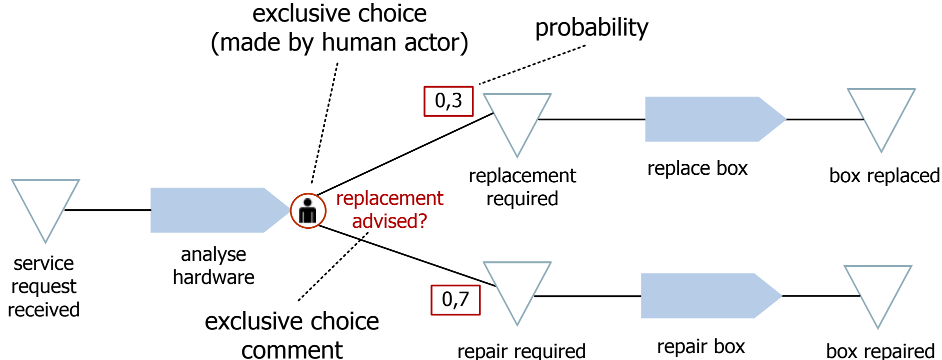


Figure 11: Example representation of two way branching

While the example in Figure 11 shows a simple exclusive choice with two alternatives only, the same concepts can be used for representing multiple selections.

3.1.3 Concurrency

If two or more parts of a business process can be executed independently, one speaks of concurrent or parallel execution. Figure 12 gives an example of splitting a business process into three paths of parallel execution. Concurrency emphasises independent execution, which include the possibility of simul-

taneous execution. Note that we do not differentiate between concurrent and simultaneous execution because we assume that usually there will be no need for enforcing true simultaneous execution.

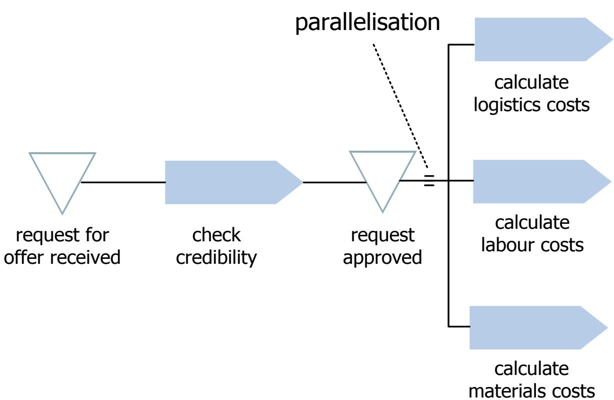


Figure 12: Representation of parallel execution

Abstract syntax: Concurrent paths of execution are started with a concurrency split. A concurrency split is associated to the event that triggers the various concurrent paths of execution. Each path of execution starts with the event that precedes the concurrent split. Each path of execution is treated like any path of execution, i.e. it may contain exclusive choices, further concurrent paths etc.

Semantics: Parallel paths are executed independently. They will often be executed simultaneously. But this is not mandatory.

3.1.4 Synchronisation of concurrent paths of execution

Concurrency is motivated by the quest for faster throughput. For this purpose a complex task is divided into a set of less complex tasks which can be performed in parallel. Usually, this division of labour implies to somehow integrate the produced results or – in other words – to synchronise the corresponding paths of execution. In order to develop an appropriate conception of synchronisation, we shall first look at possible business process structures without synchronisation.

The following statements characterise the structure and the constraints that apply to synchronising concurrent parts of a business process that does not include any synchronised parts. They are the foundation for the subsequent description of the abstract syntax and semantics. Some of them describe constraints that can be checked ad-hoc. Ad-hoc consistency comprises permissible connections of two symbols independent at a certain point in time, while ex-post consistency comprises checking for syntactical correctness of an entire graph (process model). For instance: At no point in time it is possible to connect to subprocess symbols (neither ad-hoc nor ex-post consistent). It is possible to add a concurrency split ("fork") to an event without hurting ad-hoc consistency. However, a fork without a corresponding synchroniser would hurt ex post consistency.

1. The control structure of a business process model can be regarded as a directed *tree*. The tree consists of four types of nodes: subprocesses, events, alternative choice splits ("branchings"), concurrency splits ("forks") and synchronisers.

2. The root node of a business process model is a start event.
3. A business process model includes one to n terminal events.
4. A terminal event does not trigger any further steps.
5. Each event other than a terminal event can produce one subsequent subprocess (sequence), a fork, or one or more synchronisers.
6. A process can result in one event (sequence) or in an alternative choice split.
7. An alternative choice split in turn produces n mutually exclusive events.
8. A concurrency split results in n independent (concurrent) processes.
9. Each node can be the root node of a further tree.
10. All (sub) trees that are produced by a concurrency split are called concurrent trees. This includes further subtrees that are produced by further concurrency splits. If, e.g., a concurrency split produced two subtrees t_1 and t_2 and t_1 produces a further concurrency split which results in the subtrees t_{1a} and t_{1b}, then t_{1a} and t_{1b} are concurrent to t_2.
11. The (sub) trees that are produced by an alternative choice split are called alternative trees. This includes further subtrees that are produced by an alternative choice split. If, e.g., an alternative choice split produced two subtrees t_1 and t_2 and t_1 produces a further alternative choice split which results in the subtrees t_{1a} and t_{1b}, then t_{1a} and t_{1b} are alternative to t_2.
12. If two trees t_1 and t_2 are concurrent and t_2 is split into two alternative trees t_{2a} and t_{2b}, then t_{2a} and t_{2b} are *potentially concurrent* to t_1, i.e. alternative trees within two concurrent trees are potentially concurrent. In the simplest case, an alternative tree consists of a (final) event only. The final events of potentially concurrent trees are called potentially concurrent events. The final events of concurrent trees are called concurrent events.
13. Two trees that are concurrent can be synchronised by associating their final concurrent events with a synchroniser.
14. A synchroniser is either conjunctional, i.e. it is fired only after all synchronised events have fired, or disjunctional. In the latter case, the synchroniser is fired as soon as the first of the synchronised events has fired.
15. Two potentially concurrent events can be synchronised by associating their final potentially concurrent events with a synchroniser.
16. Two trees that are not concurrent or potentially concurrent cannot be synchronised. Two alternative trees are not concurrent.
17. Two synchronisers s_1 and s_2 are mutually exclusive, if they the occurrence of s_1 excludes the possibility that s_2 may occur, too. Hence, they are not concurrent.
18. Two synchronisers are concurrent, if they may occur concurrently.
19. The synchronisation of potentially concurrent trees must finally result in more than one synchroniser. Otherwise, there would be a deadlock in the case of a conjunctional synchroniser – an event cannot occur together with the event that represents its negation. In the case of only one disjunctional synchroniser, the previous alternative choices would be meaningless, which would result in a non-sense flow of control.

20. The two or more synchronisers of potentially concurrent trees must clearly represent concurrent events or alternative choices. Otherwise, an ambiguous situation would result: two synchronisers could fire concurrently, but also mutually exclusive (see examples in Figure 19).

21. The set of synchronisers should be complete in the sense that all constellations of events are accounted for. If that was not the case, certain constellations could not be synchronised, which would imply a deadlock. The two conjunctional synchronisers used in the example in Figure 18 would not fire, if one of the associated events did not fire – which of course might happen. For an exception of this rule, see *synchronisation exception* in 4.

22. Only concurrent synchronisers can be further synchronised. They are synchronised like concurrent events.

23. Mutually exclusive synchronisers correspond to an exclusive choice. Each one of a set of mutually exclusive synchronisers produces a sequential path of execution, which are mutually exclusive.

24. If two concurrent trees are synchronised, the resulting structure is a directed graph that is terminated by synchronisers. With respect to further synchronisations, it can be treated as a tree, the final events of which are – in part – synchronisers.

The example, incomplete business process model in Figure 13 illustrates the statements above. Trees t_1 and t_2 are alternative trees. All subtrees of t_1 and of t_2 are alternative trees, too (9). Hence, a subtree of t_1 must not be synchronised with a subtree of t_2 (14). The same applies to the alternative trees $t_{2.2.1}$ and $t_{2.2.1}$. The trees $t_{2.2.2.1}$ and $t_{2.2.2.2}$ are concurrent, their final events are potentially concurrent (10). Hence, they can be synchronised, e.g. by synchronising event E5 and E7 as well as E6 and E8. The two subtrees of $t_{2.1}$, resulting in the events E0 and E1, are concurrent. Therefore, they can be synchronised. While $t_{2.1}$ and $t_{2.2}$ are concurrent, the two alternative trees $t_{2.2.1}$ and $t_{2.2.2}$ are potentially concurrent to $t_{2.1}$ (10). Hence, the concurrent subtrees of $t_{2.2.2}$, $t_{2.2.2.1}$ and $t_{2.2.2.2}$, are potentially concurrent to $t_{2.1}$ (10). This implies that they can be synchronised with $t_{2.2.2}$, e.g. by synchronising E5 and E2. Figure 14 shows an example of conjunctional synchronisation.

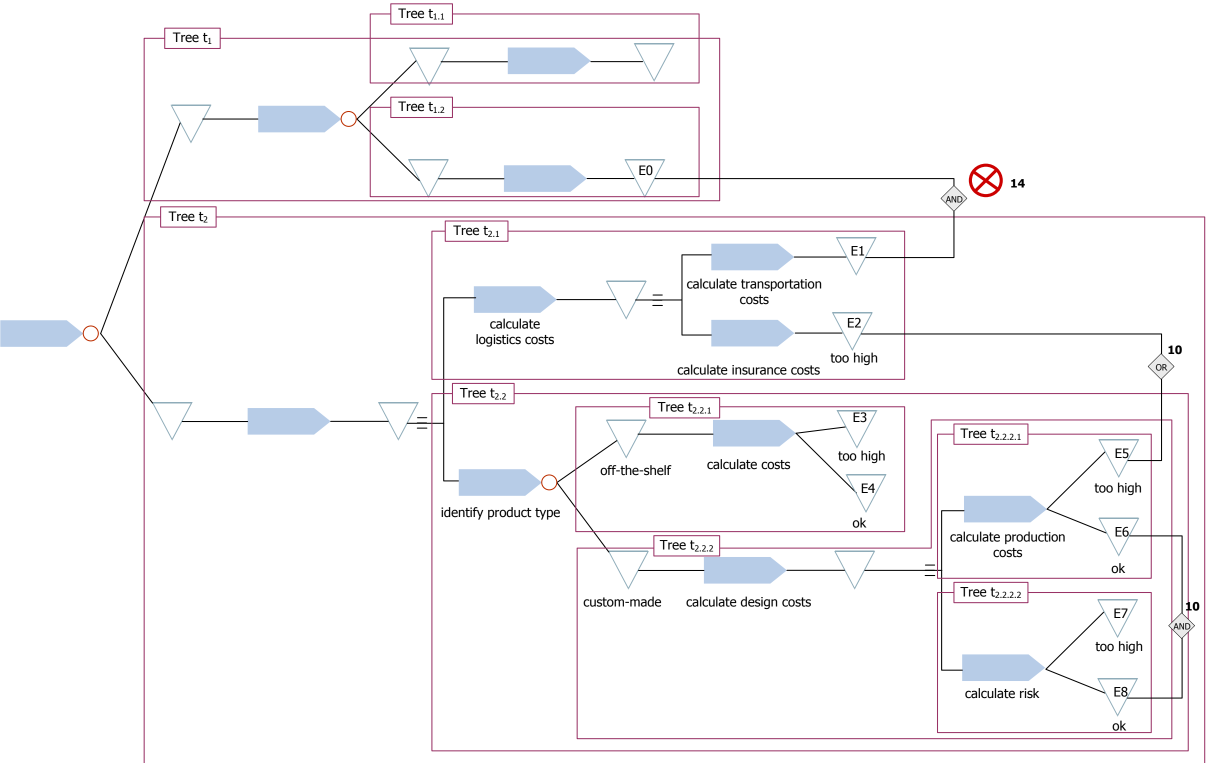


Figure 13: Illustration of synchronisation rules

Abstract syntax: The final events (or synchronisers) of concurrent or potentially concurrent trees (or graphs) are associated with a synchroniser, which can either be conjunctional or disjunctional. Each synchroniser is associated with two to many final events (synchronisers) of concurrent trees (graphs). The final events must not be part of trees that originate in the same concurrency split, i.e. that are on the same level of parallelisation. It is only required that they are concurrent (see example in Figure 15). A synchroniser must not be associated with two events that are not concurrent. Two or more synchronisers can be associated with a further synchroniser (see example in Figure 16 which is equivalent to the example in Figure 15), if they are concurrent.

Semantics: Conjunctional synchronisation means that the process that is triggered by the synchroniser is started only after all parallel paths have terminated. disjunctional synchronisation means that the subsequent process starts after the first of the parallel processes has terminated. In the latter case it is assumed – but not required – that all other processes will be terminated when the synchronisation occurs. Note, however, that this is not necessarily a case of an exclusive OR (XOR). Instead, it is possible that more than one path terminate at the same time. A synchroniser can be interpreted as an event that depends on the occurrence of other events. The set of synchronisers used for a set of concurrent trees (graphs) must be consistent, i.e. it must not be possible that they produce alternative choices and concurrent paths. Also, it must not allow for deadlocks, i.e. for constellations of events that would not fire any synchroniser.

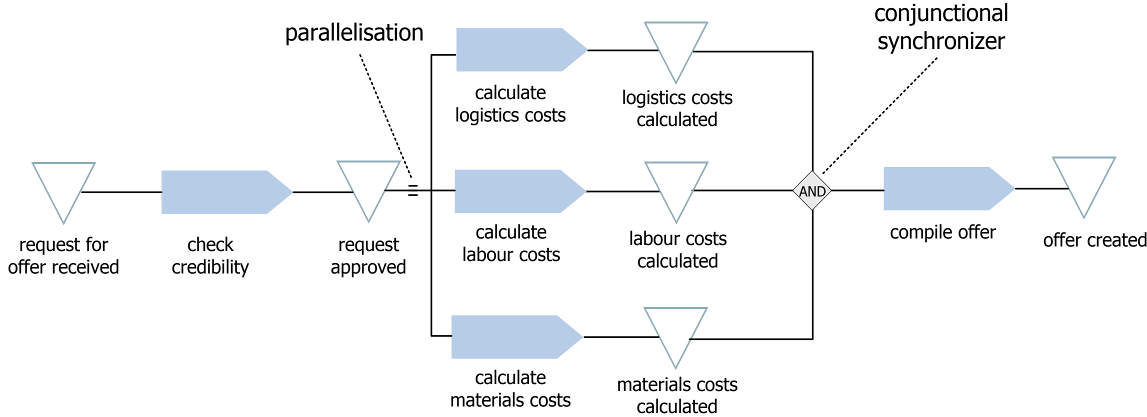


Figure 14: Conjunctional synchronisation of parallel paths

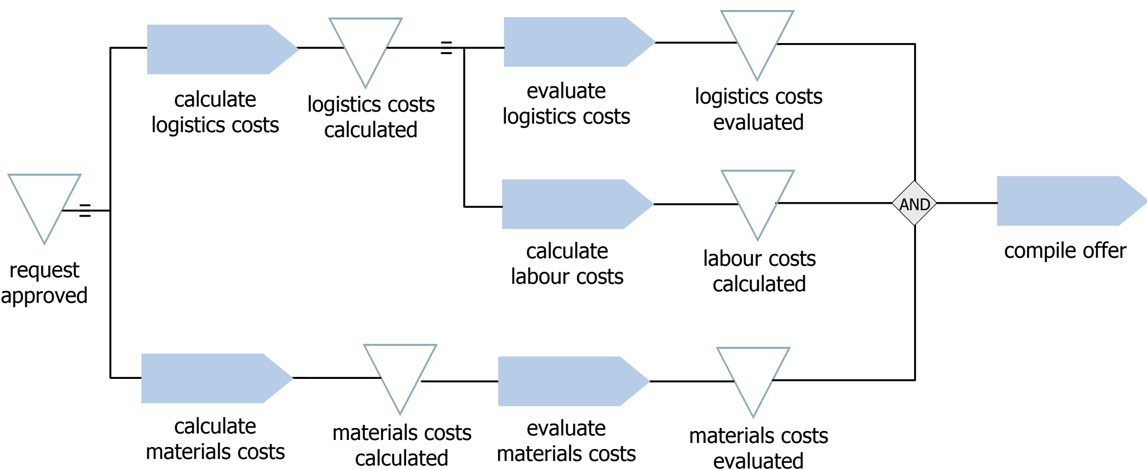


Figure 15: Conjunctional synchronisation of different levels of parallelisation

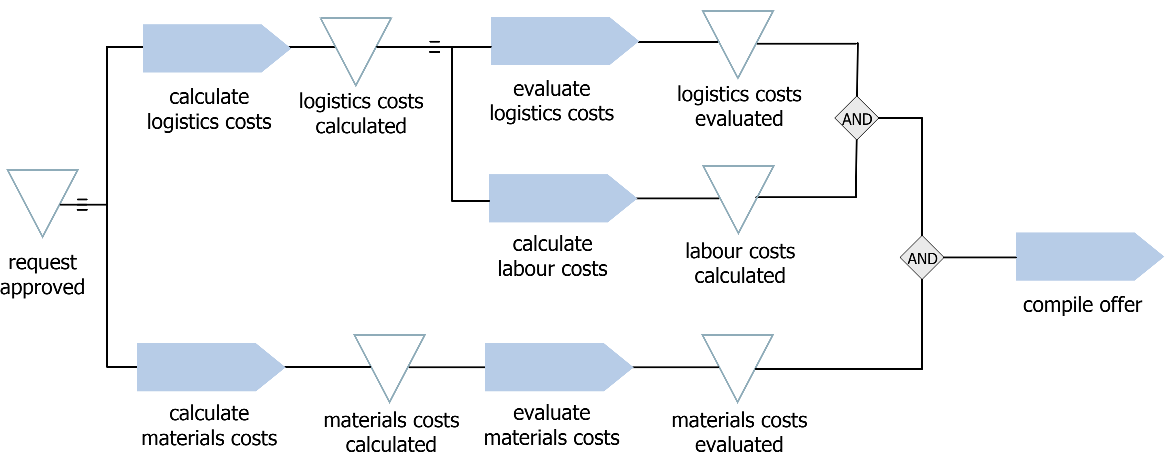


Figure 16: Synchronisation of synchronised parallel paths

Concurrent paths of execution can be synchronised by overlapping conjunctional and disjunctional compositions of terminating events. In this case, the processes that are triggered by the synchronisers represent alternative paths of execution. The corresponding notation shown in Figure 17 does not re-

veal the exclusive choice at first sight. However, analysing the both synchronisers shows that they represent mutually exclusive sets of events.

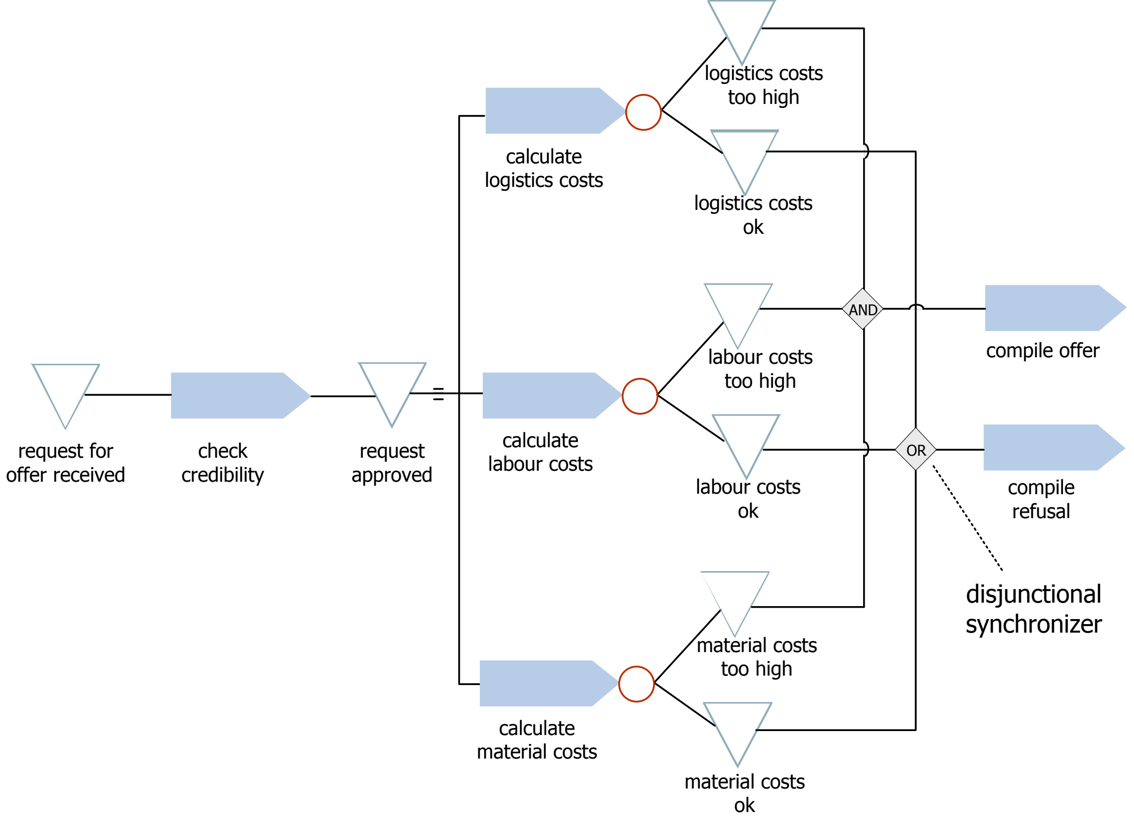


Figure 17: Combination of conjunctional and disjunctional synchronisers

Synchronisation does not only require accounting for syntactical rules. It also requires caring for consistent semantics. The example in Figure 18 shows an irregular synchronisation, because it may result in a deadlock, where none of the two synchronisers fires (see statement 19 above).

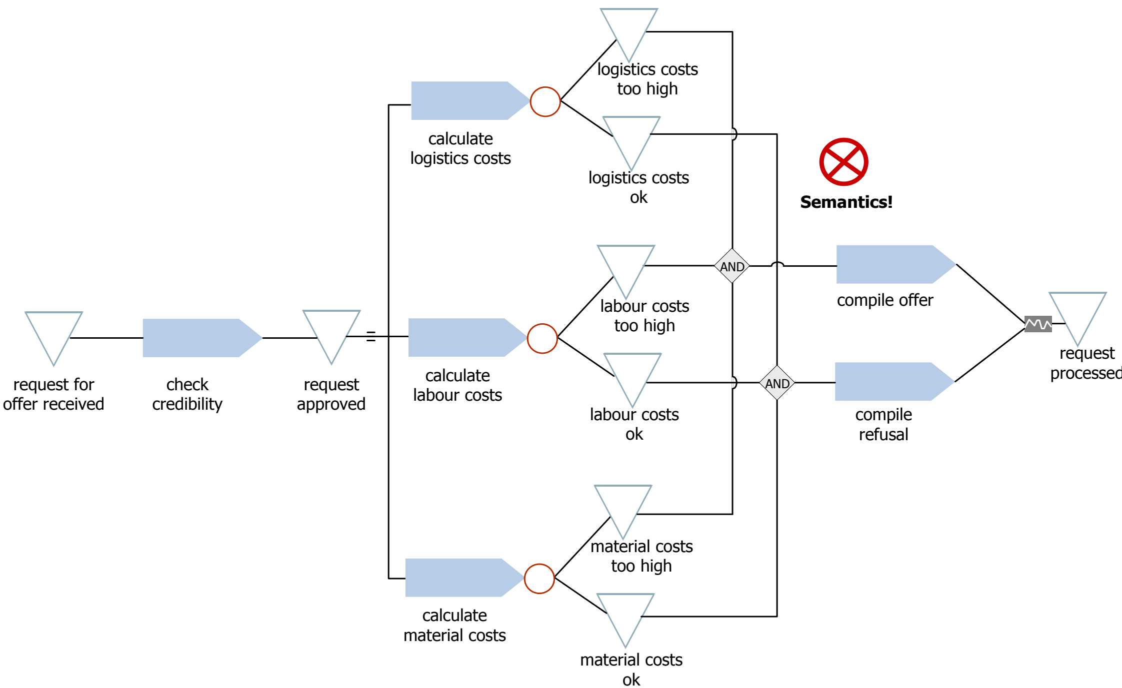


Figure 18: Irregular combination of conjunctional synchronisers

Figure 19 shows an irregular case of synchronisation, too. It may result in two mutually exclusive synchronisations. If, e.g., all events that are synchronised by one synchroniser fire, while the remaining events – which are synchronised by the other synchroniser – do not fire, then both OR synchronisations represent two non-overlapping, mutually exclusive sets. If, however, there is one event in both sets of synchronised events that fires, then both synchronisers fire concurrently. Hence, this kind of synchronisation would describe two possible flows of control that are contradictory (see statement 18 above). Note that mergers, which are used to merge to alternative paths of execution will be introduced in 3.2.1.

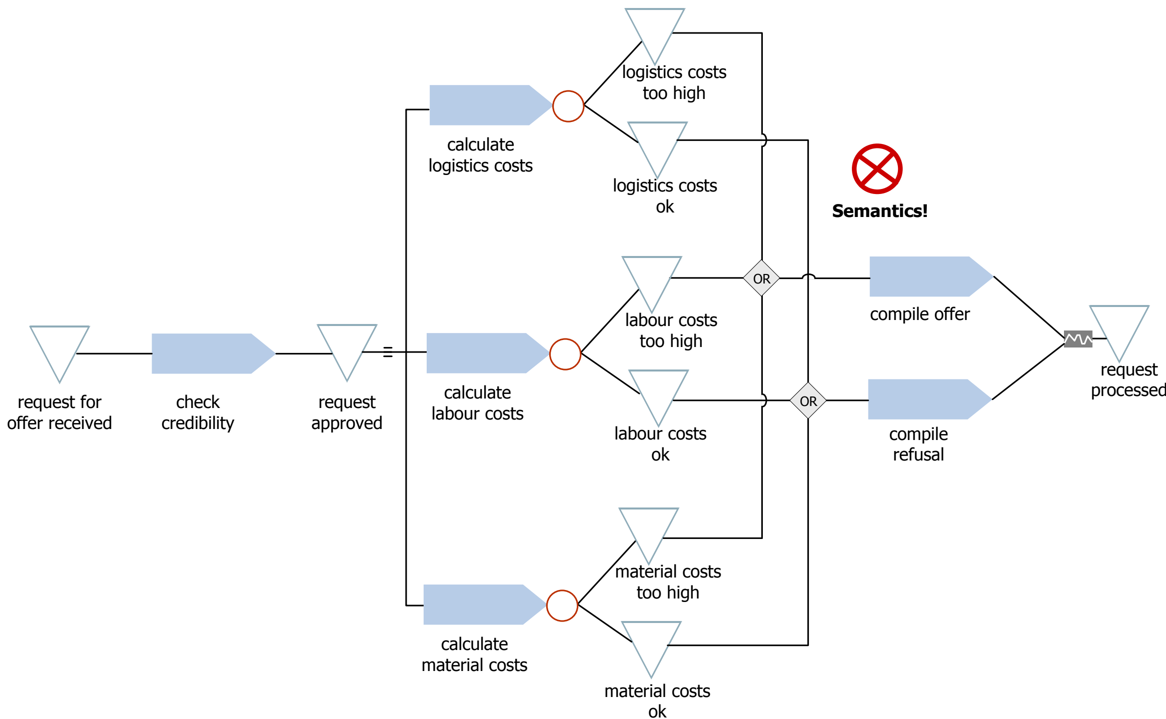


Figure 19: Irregular combination of disjunctional synchronisers

3.2 Modelling Shortcuts

To foster ease of use, the specification of a language should emphasise lean design, i.e. it should avoid conceptual redundancy. However, sometimes conceptual redundancy may promote ease of use and modelling productivity. This is the case, if frequent patterns of basic modelling concepts are replaced by higher level conceptual "shortcuts" that represent the same semantics. Shortcuts also foster a higher level of abstraction and can contribute to model integrity, because users do not have to bother with constructing models from lower level concepts. These shortcuts include merging of alternative branches of execution and iterations. We also subsume aggregations of processes under shortcuts. Note, however, that aggregations of processes are not exactly shortcuts because they are not semantically equivalent to the processes they represent in the sense that they lack information.

3.2.1 Merging of Alternative Branches

An alternative choice splits the flow of control into two or more mutually exclusive paths of execution. Sometimes these paths will eventually produce the same events, which in turn trigger the same subsequent processes. In order to avoid the visual complexity produced by multiple copies of identical flows of control, it is possible to merge alternative paths into one common path of control. This is accomplished by using the concept of a merger. The syntax of a merge is simple: the representation of two alternative trees (graphs) that are identical can be replaced by one instance of the tree. The two nodes, the identical trees start with, are the starting node of the merged tree. A starting node can be an event, a process, an exclusive choice split or a concurrency split. Figure 20 shows an example where

the two identical trees start with the same event. Merging is sometimes referred to as synchronisation of alternative branches. This, however, is misleading: It is simply an alternative – less extensive – representation.

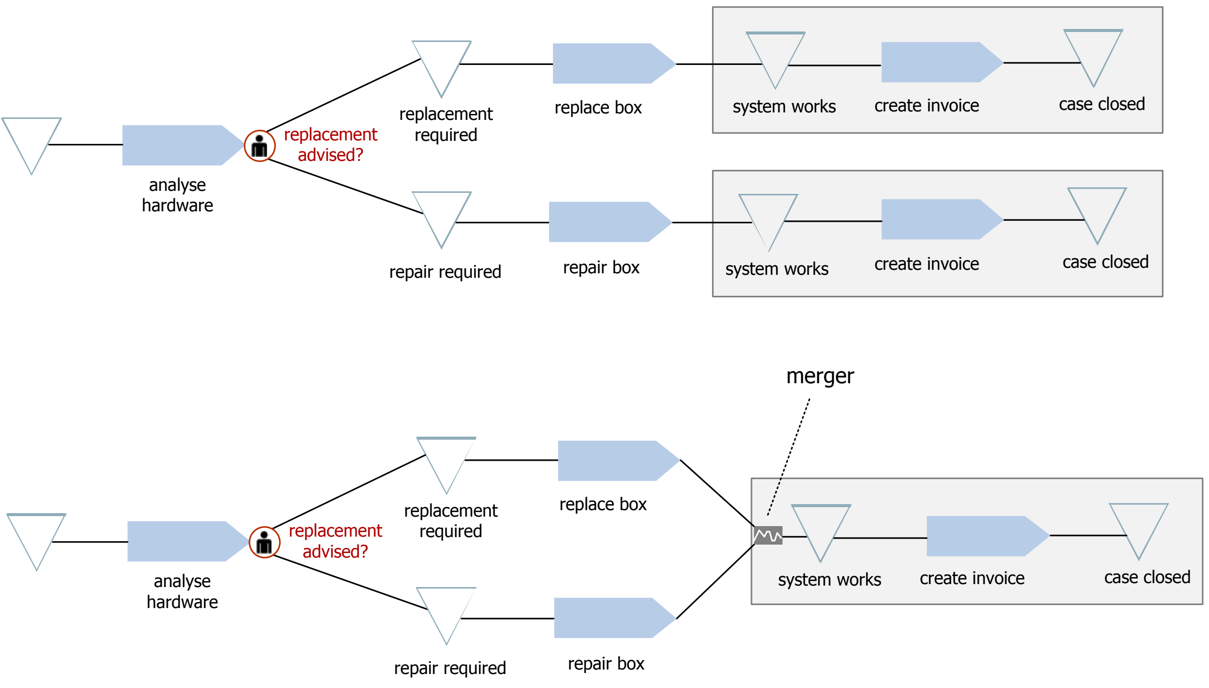


Figure 20: Merging of two alternative paths, starting with common event

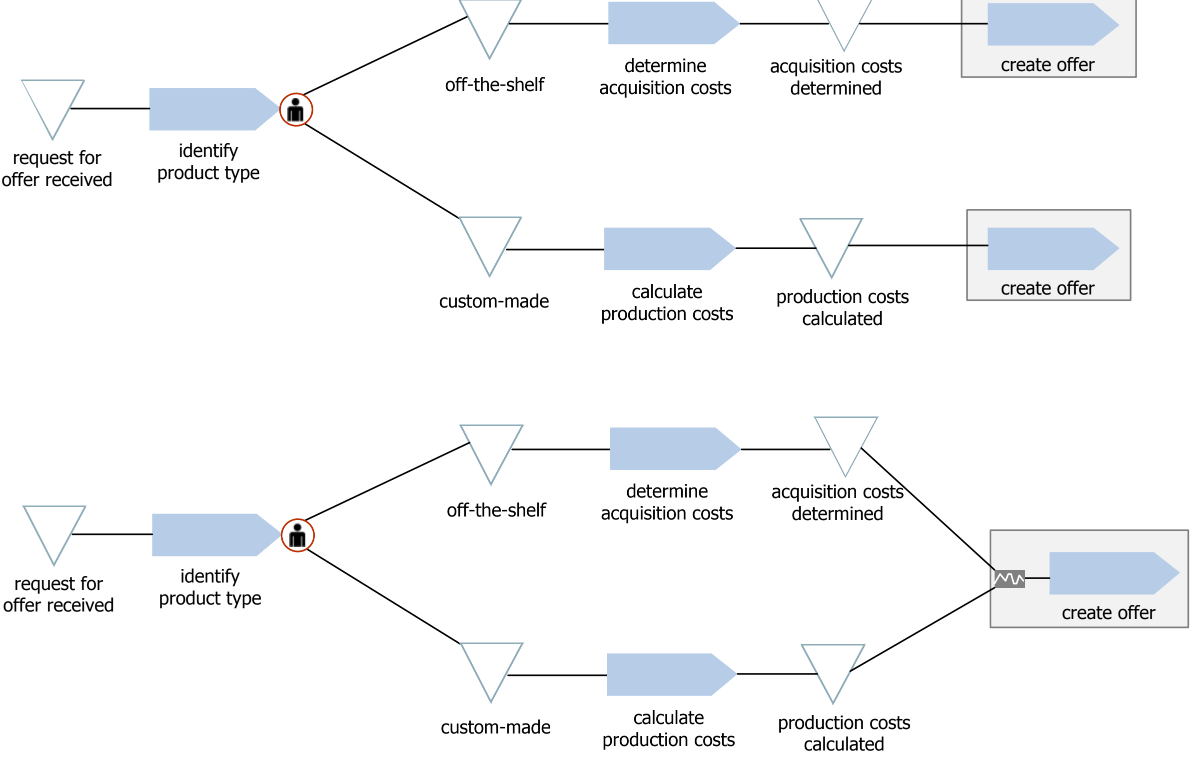
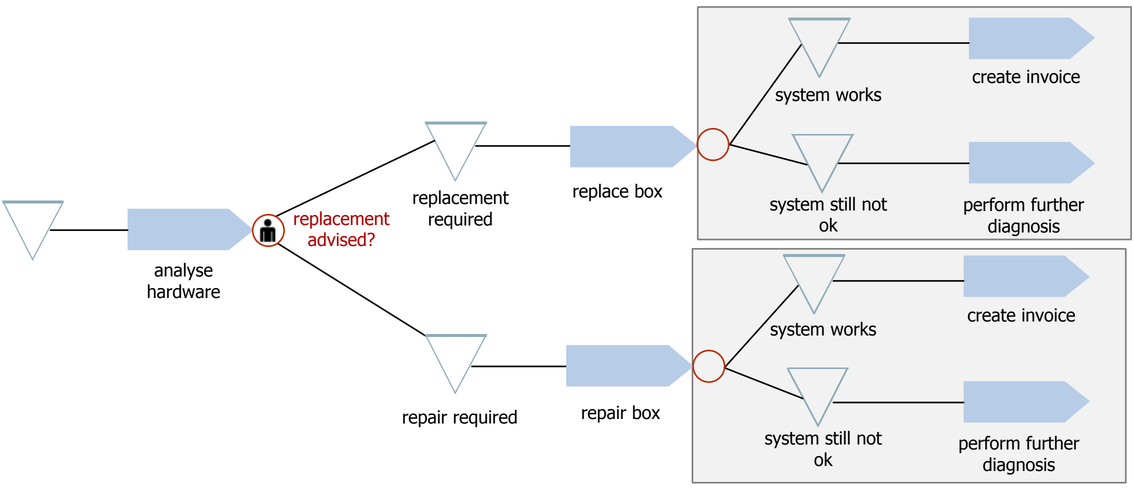


Figure 21: Merging of two alternative paths, starting with common process

We assume that in most cases an event makes more sense. It seems more likely that two different processes trigger the same event than that two different events trigger the same process. However, since merging does not change the semantics of a business process model, we also allow for merged paths starting with a common process. The example in Figure 21 illustrates that it is advised to thoroughly check whether the starting process is really the same in both alternative paths.

Figure 22 shows an example where the first node of the merged trees is an exclusive choice.



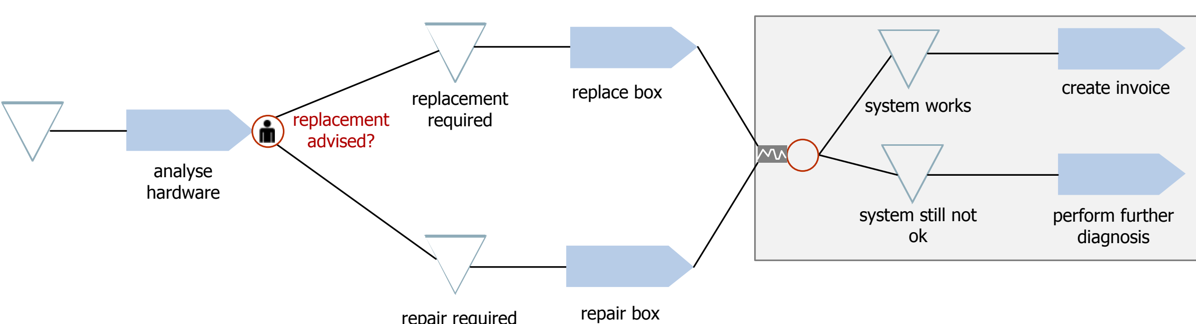


Figure 22: Merging two exclusive choices

Figure 23 illustrates that according to the general rule that defines merging, it is also possible to partially merge two alternative exclusive choices. However, while this is syntactically correct and the semantics are unambiguous, it should be thoroughly considered whether it is appropriate using this kind of merge, since it might be perceived as confusing.

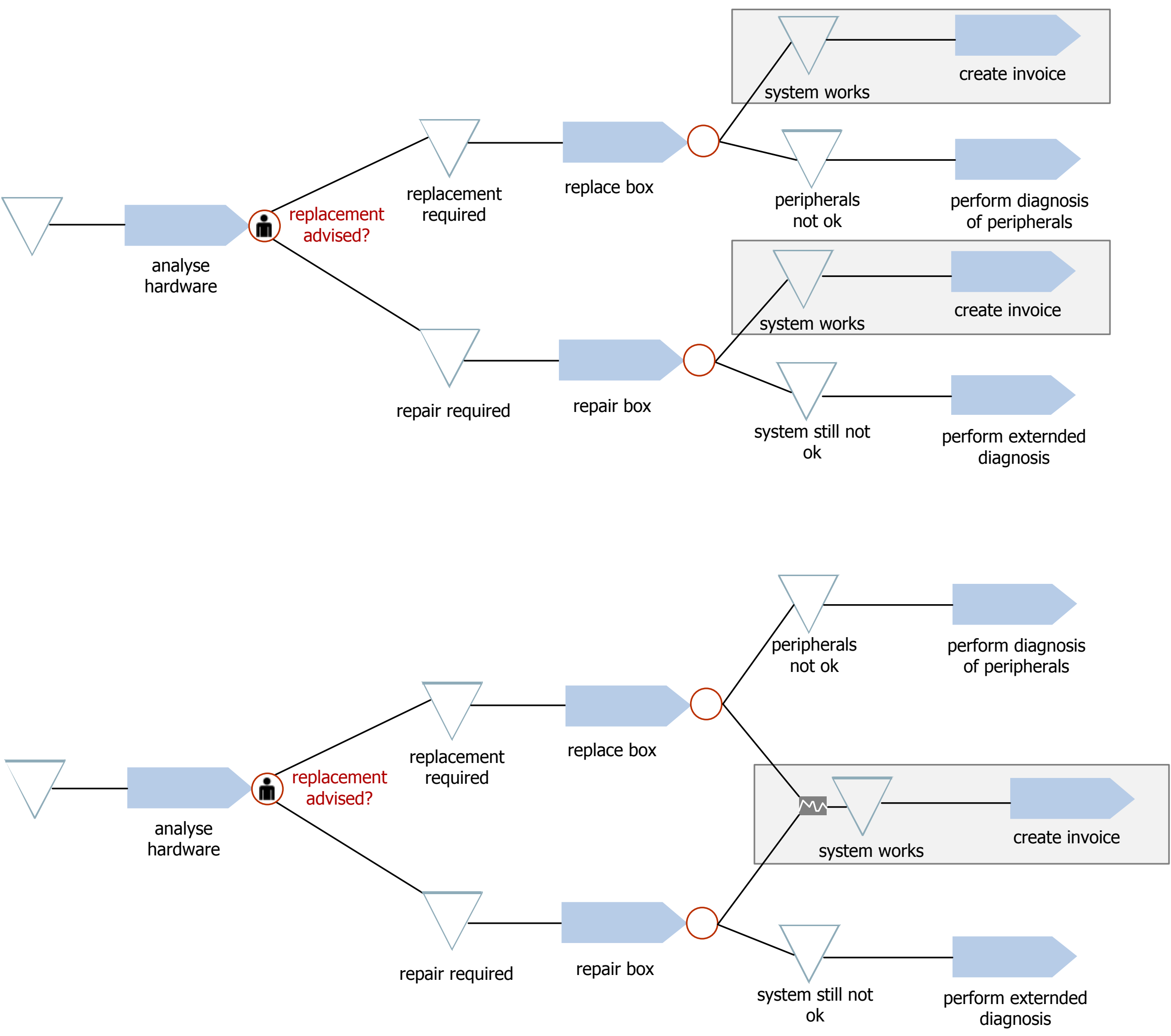


Figure 23: Partial merging of two alternative exclusive choices

Abstract syntax: Merging results in replacing n identical alternative trees (graphs) by one instance of the tree (graph). The identical trees may start with a common node, which may be an event, a process, an exclusive choice split or a concurrency split. To indicate the start of a merge, a merger is associated to the starting node. It is not permitted to merge paths that are part of concurrent trees.

Semantics: Merging alternative paths of execution is not a control structure. It simply allows for representing multiple identical paths of execution by one path only.

3.2.2 Aggregate Subprocess

The complexity of large business process models may compromise their comprehensibility because they overcharge users with too much detail. Furthermore, it may be the case that a representation medium, e.g. a computer screen, does not allow for presenting an entire business process model in a readable size. To cope with this problem it is a common approach to make use of composition and decomposition. Instead of composition we speak of aggregation: It is possible to aggregate a set of

processes into an aggregate process. While an aggregation of this kind is mainly a replacement of a number of symbols by one, it nevertheless requires some considerations with respect to its syntax. On the one hand, it needs to be defined how a sequence of subprocesses can be grouped into an aggregate process. Second, there need to be guidelines for how to aggregate the features (e.g. level of automation, associations to organisational units and resources) of the respective subprocesses.

An aggregate process comprises a sequence of subprocesses. It is triggered by the event that triggers the first subprocess and terminates with the events that result from the final subprocess. The sequence may comprise completed (synchronised) parallel executions (see Figure 26). It may als include branchings – either as an outcome of the final subprocess or branchings that have been merged to a common sequence. An aggregate process cannot be assigned a type (level of automation, internal/external), nor can it be assigned organisational units or other resources. This is for two interrelated reasons. An aggregate process does not have a type on its own. Its semantics depends entirely on the subprocesses it is comprised of. Therefore, it is not appropriate to explicitly assign a type. Secondly, the subprocesses that form an aggregate process may be of different types. In this case, it would not be possible to derive an appropriate "type" of the aggregate process. In a particular implementation, one may decide that the modelling tool assigns a type in cases where all subprocesses share the same time. The same holds for resources that are assigned to the included subprocesses. Figure 24 to Figure 27 illustrate the rules for building valid aggregate processes.

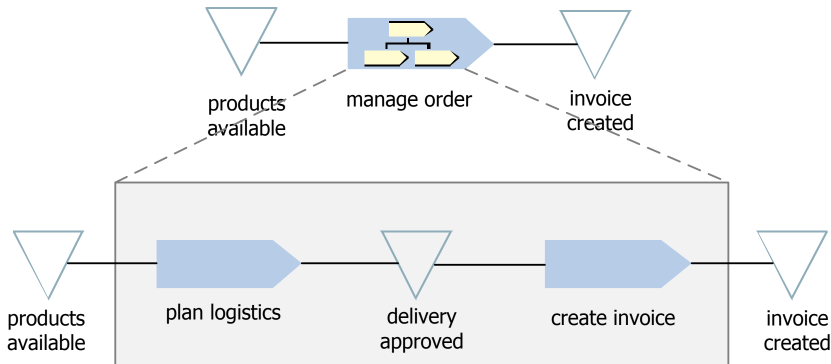


Figure 24: Aggregated process as replacement of sequence

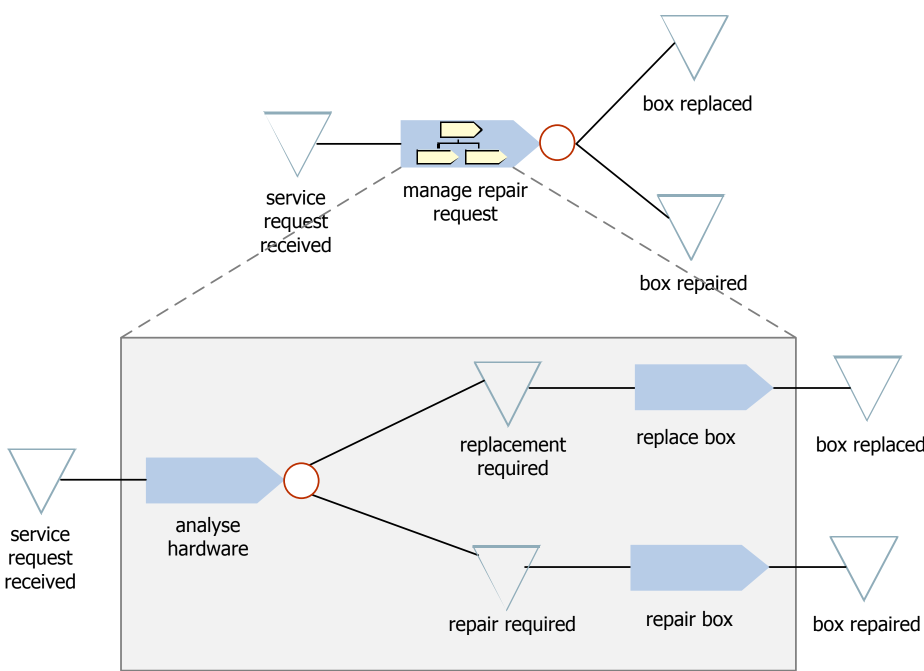


Figure 25: Representation of branching processes

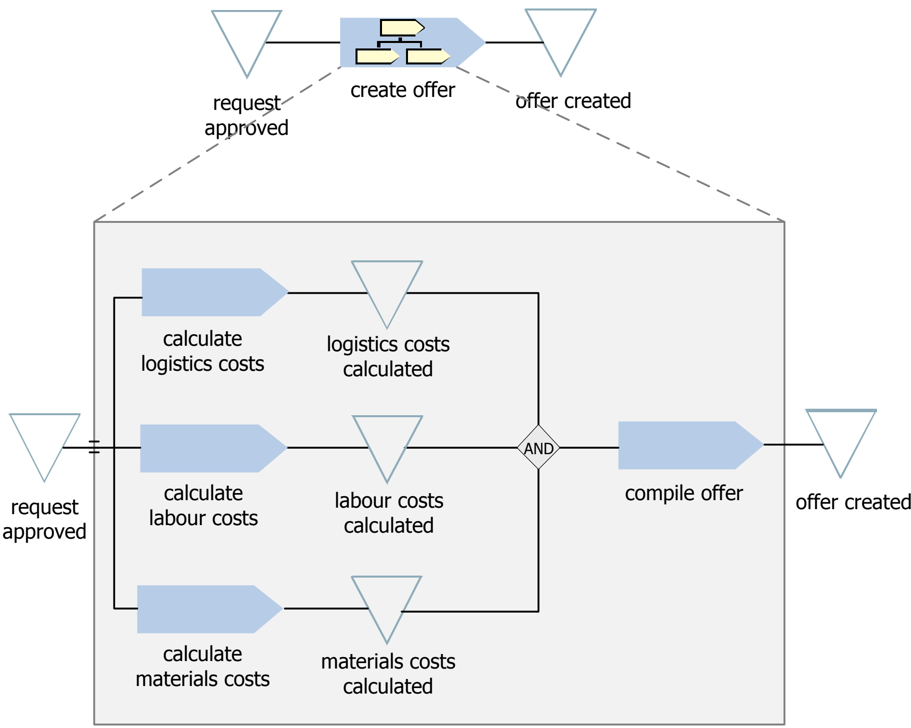


Figure 26: Replacement of parallel paths

Abstract syntax: A set of processes can be composed into an aggregated process only, if the comprised processes have one common starting event and one common terminating event or a common set of mutually alternative terminating events (exclusive choice). Hence, it is not permitted for an aggregated process to represent a set of processes that result in events which are part of concurrent paths of execution.

Semantics: The semantics of an aggregated process is simple: It is a placeholder for a part of a business process.

Figure 27 shows an example of an aggregated process that does not conform to the syntactical constraints because it does not end in one event or a set of mutually exclusive events.

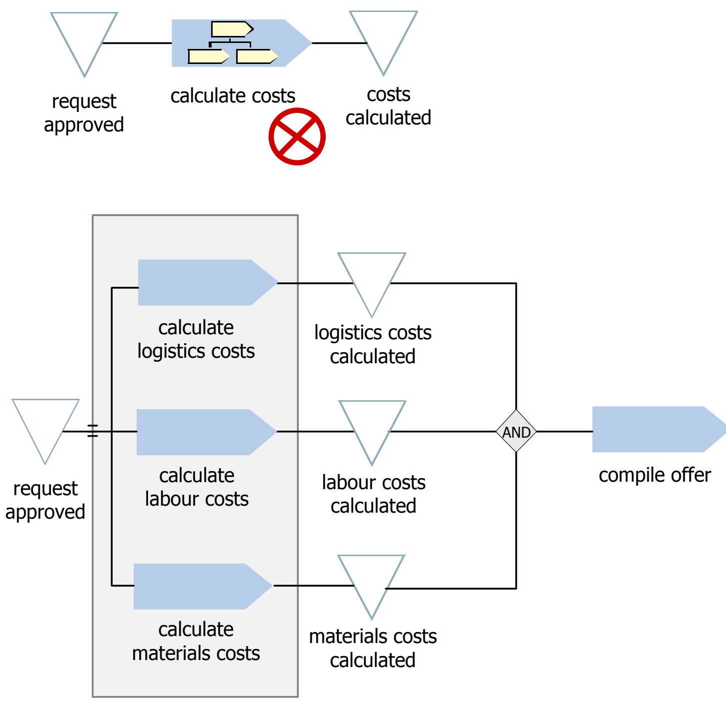


Figure 27: Example of prohibited use of aggregated process

3.2.3 Iteration

It seems that the language concepts introduced so far allow for expressing that a particular part of a business process is executed more than once in a sequential order. After the final process of the part that is subject of repetition, an exclusive choice could represent whether to go back and repeat or to continue with the subsequent process. Figure 28 shows a corresponding representation of an iteration.

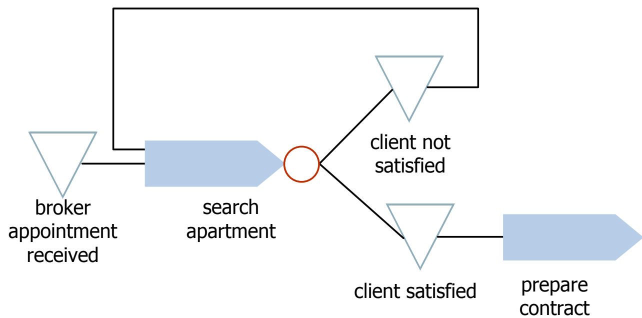


Figure 28: Representation of an iteration without specific concepts

However, this approach has a number of disadvantages. Firstly, the construction shown in Figure 28 would allow for “repeat until” control logic only. “While” and “n-times” control logic would require

further concepts. Secondly, there is a syntactical problem. Normally, a process type is triggered by one and only one event type. If an iteration would be represented as shown in Figure 28, this rule would have to be relaxed – resulting in a remarkable challenge to validate syntactical correctness: If a process type is associated with more than one preceding event, it would be required to check whether the first occurrences of the corresponding events actually happen before or after the process type. This effort would be even higher in the case of nested iterations. The example in Figure 29 illustrates the risk of confusing control structures, which is well known from "go to" statements in programming. Thirdly, it is suited to cause semantic confusion. Usually, an event type that is produced by a process type indicates that a corresponding event occurs after the corresponding process. In the case of an iteration, instances of the event type could both occur before and after instances of the process type. Fourthly – related to the first problem – it would imply a bigger effort to identify iterations within a business process. Hence, it would be more challenging to support corresponding functions within a tool.

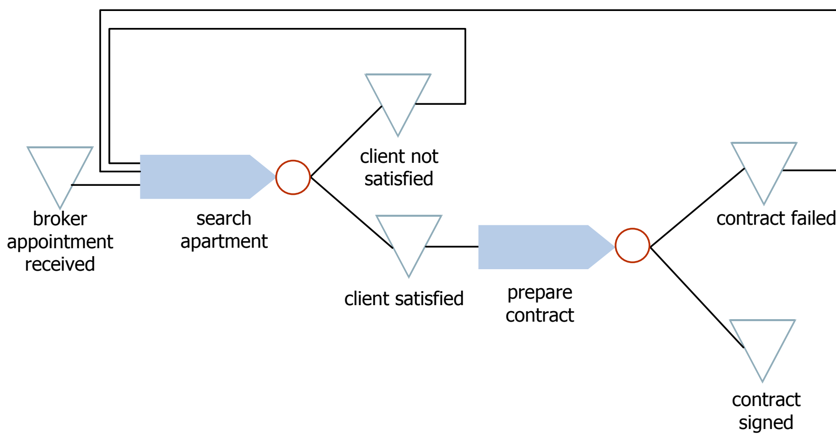


Figure 29: Representation of nested iterations without specific concepts

For these reasons, the OrgML provides specific language concepts for representing iterations. "Iteration Start" serves to indicate the beginning of that part of a business process that is subject of a repetition. We call this part *iteration body*. "Iteration End" is used to define the end of an iteration body.

There are three kinds of control logics to define the conditions that specify an iteration. To enable "while" control logic the language provides a corresponding symbol that is associated with a guarding condition. This symbol represents the iteration start. The guarding condition corresponds to the event that precedes the iteration start symbol. A further symbol is used to depict the iteration end. Note that it is not possible to combine events to a more complex condition. If this is required, it would be necessary to represent this complex condition by one event. A "repeat until" loop is represented by using a symbol to depict the iteration start and an "until" symbol that represents the iteration end. The terminating condition is represented by a set of mutually exclusive events that may result from the iteration. The same structure is used for realising "n-times" control logics. In this case, the "until" symbol is replaced by a symbol that shows a number which indicates the number of iterations. This number may be represented by a constant or – more likely – by referring to some variable which could e.g. be provided by a service of an associated class diagram. Figure 30 shows an example of an iteration that is controlled by "repeat until" logics. While the example is restricted to one process that is

repeated, it is possible to define an iteration for any part of a business process – provided certain syntactical rules are not hurt (see below).

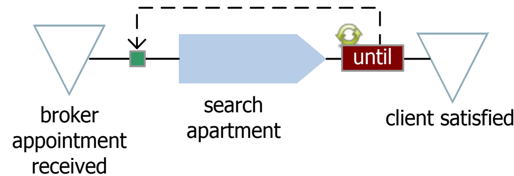


Figure 30: Example of an iteration with "repeat until" logics

An iteration may be terminated alternatively by more than one event. This can be expressed by using an exclusive choice subsequent to the end of an iteration. Figure 31 illustrates this for a further "repeat until" and a "n-times" iteration. We assume that there is no need for "n-times" control structures that use increments different than one.

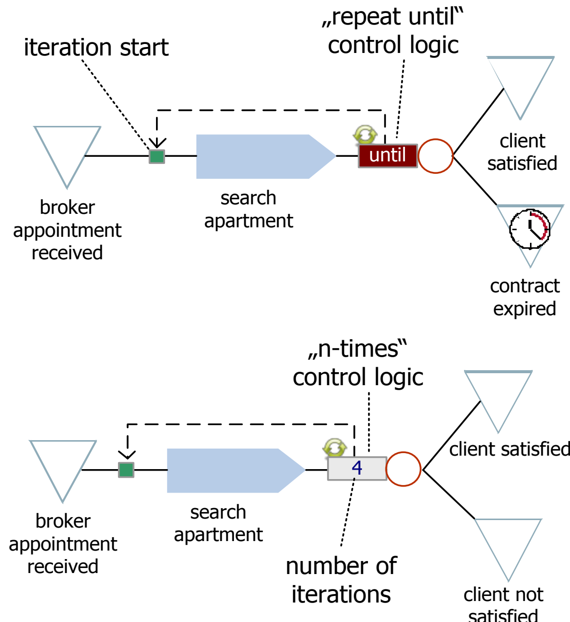


Figure 31: Examples of multiple alternative events following an iteration

The example in Figure 32 illustrates the syntax of "while" control structures. If the guarding condition if defined as a logical conjunction, all included conditions should be represented separately at the end of the iteration – because each one of them could terminate the iteration. Therefore, it should be defined what is supposed to happen in any case. To foster integrity, a modelling tool could allow for generating the guarding condition from negations of the conditions that define the subsequent events.

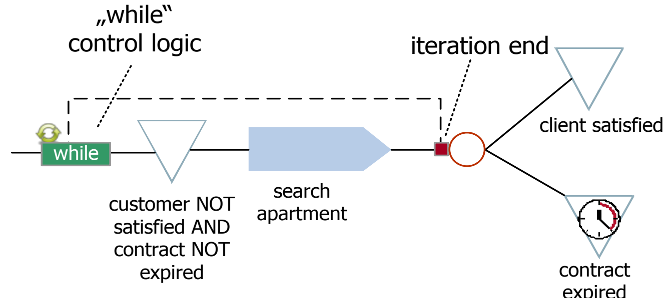


Figure 32: Example of a "while" control structure

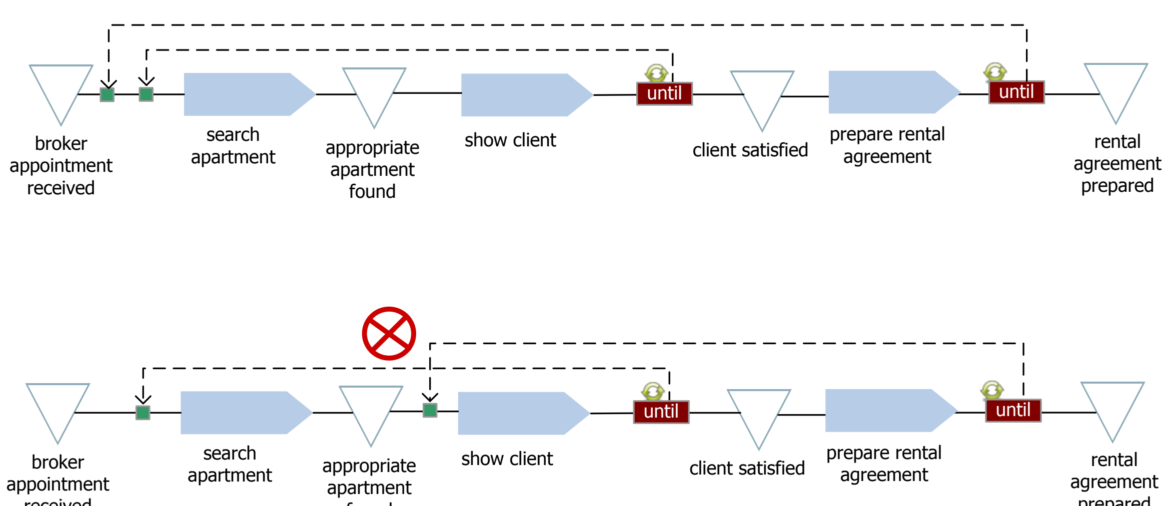


Figure 33: Nested Iterations

Iterations may include alternative branches and parallel paths of execution. This implies additional syntactical constraints. For parallel paths of execution the rules are fairly straightforward. A parallel part of a business process needs to be completely embedded into an iteration body, i.e. it needs to be synchronised before the end of the iteration body. Figure 34 shows one example that violates this rule as well as one correct representation.

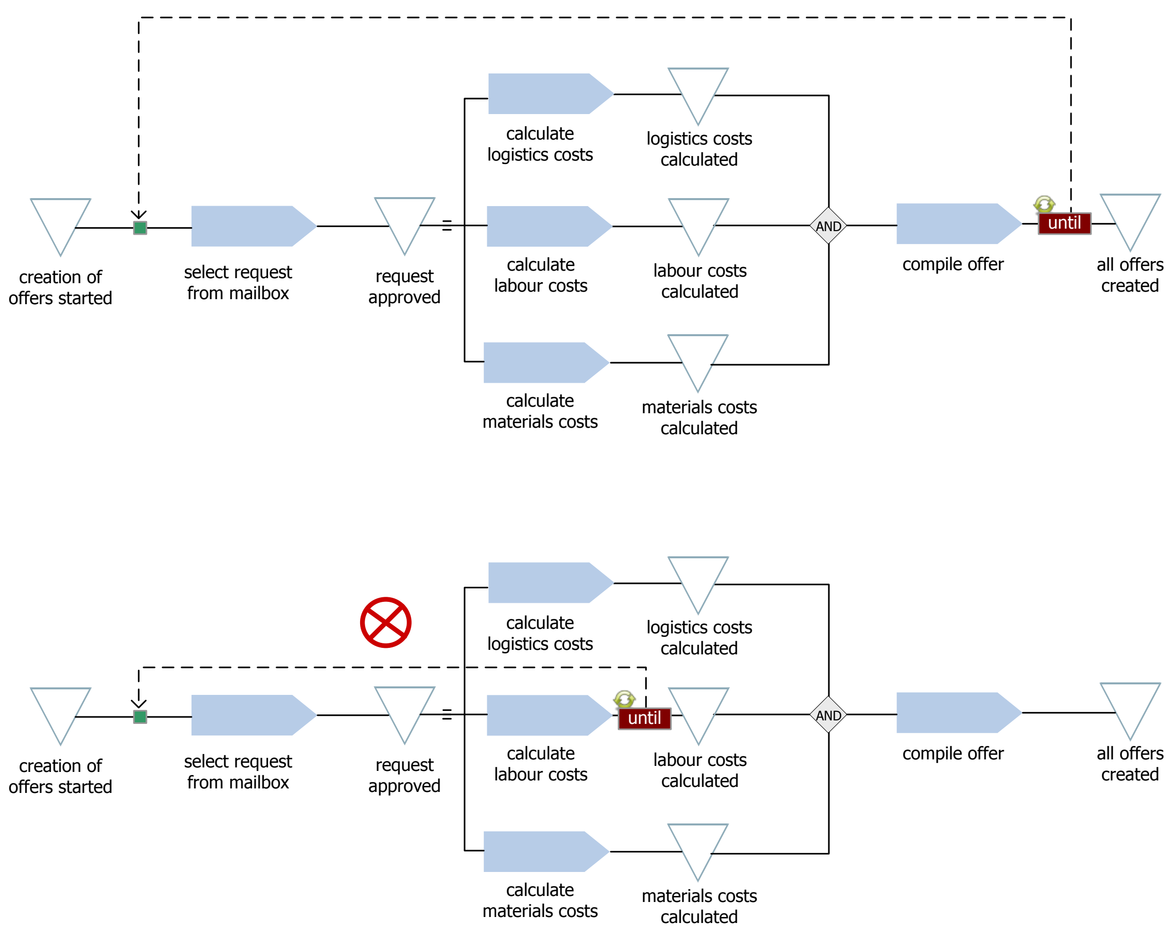


Figure 34: Parallel paths of execution within an iteration body

The representation of iterations that include exclusive choices is more challenging. While corresponding control structures are well known from program design – and are nicely supported e.g. by structograms – defining a corresponding representation for business processes recommends accounting for the fact that prospective users may not be as familiar with algorithmic structures as programmers. Therefore, the notation should emphasise comprehensibility and integrity. In general, the branches of an exclusive choice that starts within an iteration body should be merged before the iteration end. This rule can be applied for constructing nested iterations, too. Figure 35 shows a corresponding example.

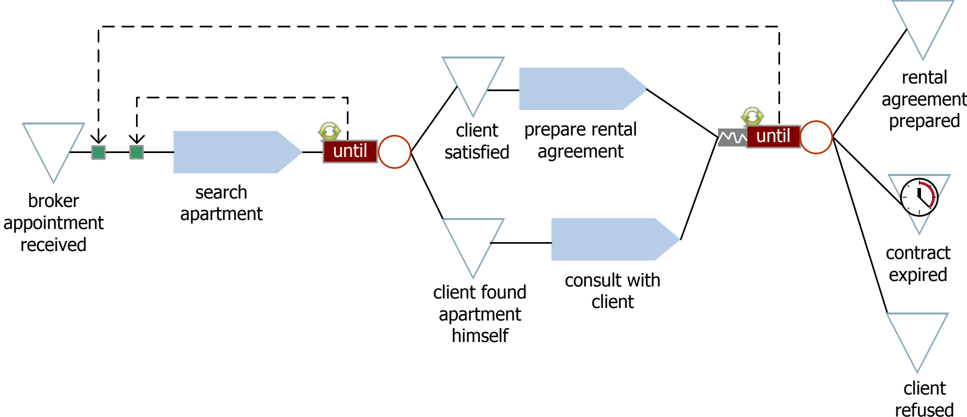


Figure 35: Nested iteration with branching

However, if one of the branches does not include a process, merging becomes a syntactical challenge. Modifying the above example we assume a business process where a broker is searching for an apartment. He continues searching until the client is satisfied or the contract he has with the client expires. If the client is satisfied with an apartment the broker prepares the rental agreement with the landlord. If the contract is subsequently concluded, the process terminates. Otherwise, it starts all over again with searching for an apartment. Figure 36 shows a possible representation that is, however, syntactically wrong. The branch that consists of one event only is not clearly included in the surrounding iteration.

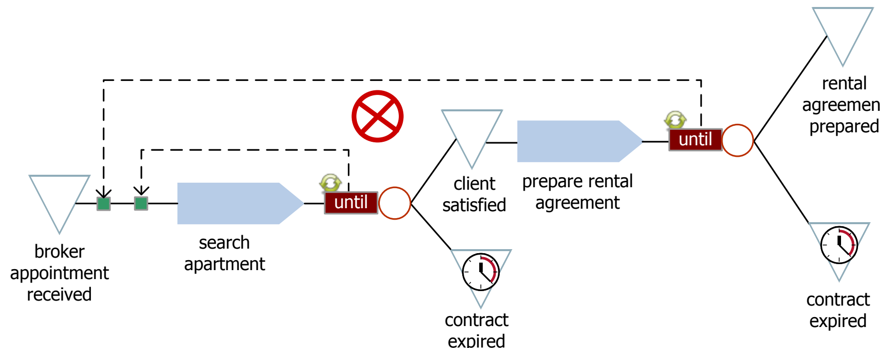


Figure 36: Syntactically wrong representation of nested iteration

Figure 37 shows the correct representation of the example. It includes a merger. However, it is not permitted to merge an event and a process. Therefore, a different concept is required. It is indicated by the dotted line that connects the event to the merger symbol. The dotted line serves to represent an “empty process” that is part of a merger.

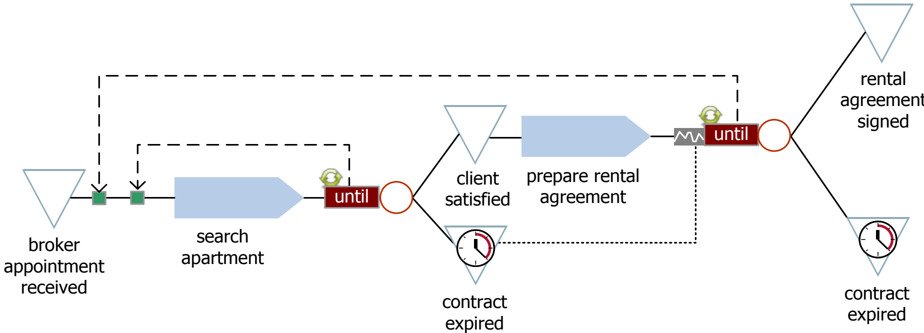


Figure 37: Correct representation of nested iteration

Note that the model in Figure 37 implies a semantic constraint: The event "contract expires" which terminates the inside iteration must also be part of the disjunction that terminates the outside iteration. Otherwise, it would be possible for the iteration not to terminate.

Abstract syntax: The beginning and the end of an iteration are defined by an iteration start and an iteration end respectively. An iteration start is preceded by exactly one event, while an iteration end is succeeded by one event or a set of mutually exclusive (XOR) events. Both, iteration start and iteration end must be part of the same overall path of execution, i.e. alternative and concurrent paths of execution must be completely and not just partly included in the iteration body. In the case of alternative branches this implies that an iteration body that starts before the branching must not be terminated within one of the branches, but only after these have been merged. If a branch does not include a process, it needs to be merged anyway. This is accomplished by a special kind of association between the event and the merger. There are three kinds of logics that serve to control iterations. A "while" structure is represented by a corresponding concept that acts as the iteration starts and it associated with the iteration end. A "repeat until" structure is represented by a corresponding concept that acts as the iteration end – and is associated with the iteration start. A "n-times" structure is represented like a "repeat until" structure, with a number that indicates the number of iterations. The number can be represented either by a constant or by a reference.

Semantics: The iteration body is repeatedly executed. The number of repetitions depends on the condition defined in the control structure. If the iteration is controlled by a "while" structure, the repetition continues as long as the event that precedes the iteration start evaluates to true. If the iteration is controlled by an "until" structure, the repetition continues until one of the events that follows the iteration end evaluates to true. If a "n-time" control structure is used, the iteration body is repeated n times.

4 Advanced Control Structures

In addition to the control structures presented so far, there are further control structures that are assumed to be used less frequently. They are not as obvious as the basic control structures. Therefore we call them advanced control structures. Note that we do not have sufficient knowledge at this point to assess the relevance of these control structures for business process modelling in practice. The actual future use of the language should help to clarify this issue.

4.1 Arbitrary Sequence

It is conceivable that a set of processes must be executed sequentially, but not in a particular order. For example: Within a clinical diagnosis a patient has to go through an audiometry (process A), he must have an X-ray (process X) and a blood sample has to be taken (process B). Since these tests do not depend on each other, they could be performed concurrently. This, however, is not possible for technical reasons. At the same time, it would not be advisable to define a particular order that has to be followed with each instance of the clinical process: Apparently, the appropriate order depends on the availability of resources (medical staff and devices), which may vary from process instance to process instance. Often, it is sufficient, if not most appropriate, to have the scheduling done by those people who operate the process. Such a constellation can be represented by the concept of an *arbitrary sequence*. It is an abstraction that represents the set of sequences that can be constructed from a set of processes. For n processes, the number of possible sequences, hence the number of permutations of n, equals the factorial of n. The selection of one of these possible sequences happens only at run-time – either by a human actor or some machine. Hence, it resembles the idea of an ad hoc workflow. It is also similar to an aggregated process in the sense that it represents a set of processes.

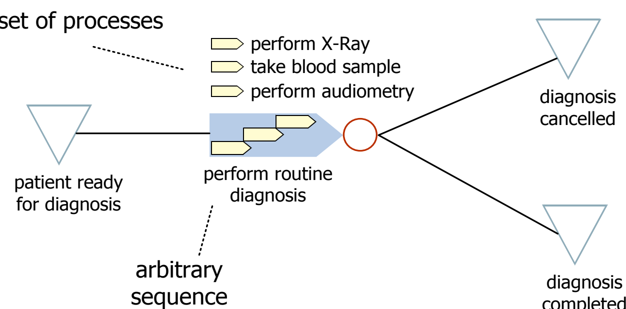


Figure 38: Example of arbitrary sequence

However, different from an aggregated process, it cannot be decomposed into a particular order – because the absence of a particular order on the type level is its main characteristic. Also, events that trigger a process within the sequence are not accounted for. This is for a good reason: The semantics of an event that may trigger a process is not independent from the particular sequence of execution. For instance: Take the event "Process A terminated". Its semantics with respect to constructing a valid sequence varies with the actual order of process A. An arbitrary sequence is represented by a special symbol that is supplemented by a list of the included processes (see Figure 38).

Abstract syntax: An arbitrary sequence is triggered by an event and produces one or more mutually exclusive events. The representation of an arbitrary sequence is supplemented by the names of the processes it represents. From a syntactical point of view it corresponds to an aggregate process. However, different from that, it cannot be decomposed.

Semantics: An arbitrary sequence represents a set of processes which have to be executed sequentially, but not in a particular order.

Note that the constellation addressed by the concept of an arbitrary sequence could also be represented in a more sophisticated way: It would be possible to define the construction of a particular sequence depending on the relevant states of a process instance and the resources it requires. This could, e.g., be accomplished by a scheduling algorithm.

4.2 Arbitrary Sequence with Partial Order

If a partial order is defined for a set of processes, we speak of an arbitrary sequence with partial order. The partial order reduces the number of possible sequences. A partial order can be defined by a set of ordering relations, e.g. process A before process D, process E before process C etc. In addition to that it is conceivable to used conditional partial orders, e.g. A before D, if E before C. Since it is not clear whether this control structure is relevant for a notable number of scenarios, we do not bother with defining language concepts to specify arbitrary partial orders. Instead, the representation of partial order is restricted the defining orders within the list of processes that supplement the specific symbol for a partial arbitrary order. An arrow pointing from process A to process B defines the partial order that A has to be executed before B (see example in Figure 39).

Abstract syntax: The abstract syntax of an arbitrary sequence with partial order corresponds to that of an arbitrary sequence. It is supplemented by directed edges (arrows) that represent partial orders between two processes

Semantics: An arbitrary sequence with partial order restricts the number of possible sequences by a set of partial orders. The partial orders must be consistent and must not restrict the number of possible sequences to one.

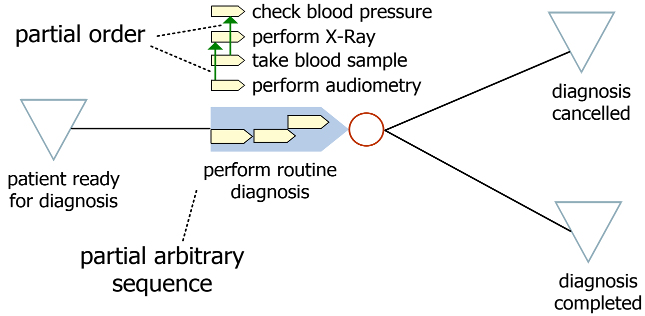


Figure 39: Example of partial arbitrary order

4.3 Synchronisation Exception

The complete synchronisation of concurrent paths of execution requires accounting for every possible constellation of final events. Some of the possible constellations may seem extremely unlikely. However, not accounting for all possible constellations bears the risk of deadlocks. A purposeful incomplete synchronisation allows for leaving out constellations that seem extremely unlikely without jeopardising the consistency of a process. This is accomplished by a special kind of exception: A *synchronisation exception* is an abstraction of virtual synchronisers that cover all constellations of final events that are not accounted for by explicit synchronisers. A synchronisation exception will fire, if all concurrent trees (graphs) terminate with a final event and the resulting pattern is not accounted for by any explicit synchroniser. Synchronising parallel paths of execution may fail, too, if a process in one of the paths does not terminate – or does not produce one of the events it is supposed to produce. Such a constellation could be regarded as a synchronisation exception as well. However, there are two reasons why this would not be appropriate. First, the problem occurs during the execution of a process. Hence, the exception should rather be assigned to the process. Second, this kind of synchronisation exception would be hard to detect: How would one decide whether a concurrent path of execution that has not yet produced a final event will fail to do so? The concept of a synchronisation exception should be used with great care: It makes sense only, if certain synchronisation patterns are extremely unlikely *and* modelling them explicitly would clearly jeopardise the comprehensibility of the synchronisation structure. Furthermore, the constellations that are left out should be similar in the sense that they can all be treated with the same kind of exception handling. Figure 40 illustrates the use of a synchronisation exception. The red dots illustrate one possible constellation that is not covered by the regular synchronisers.

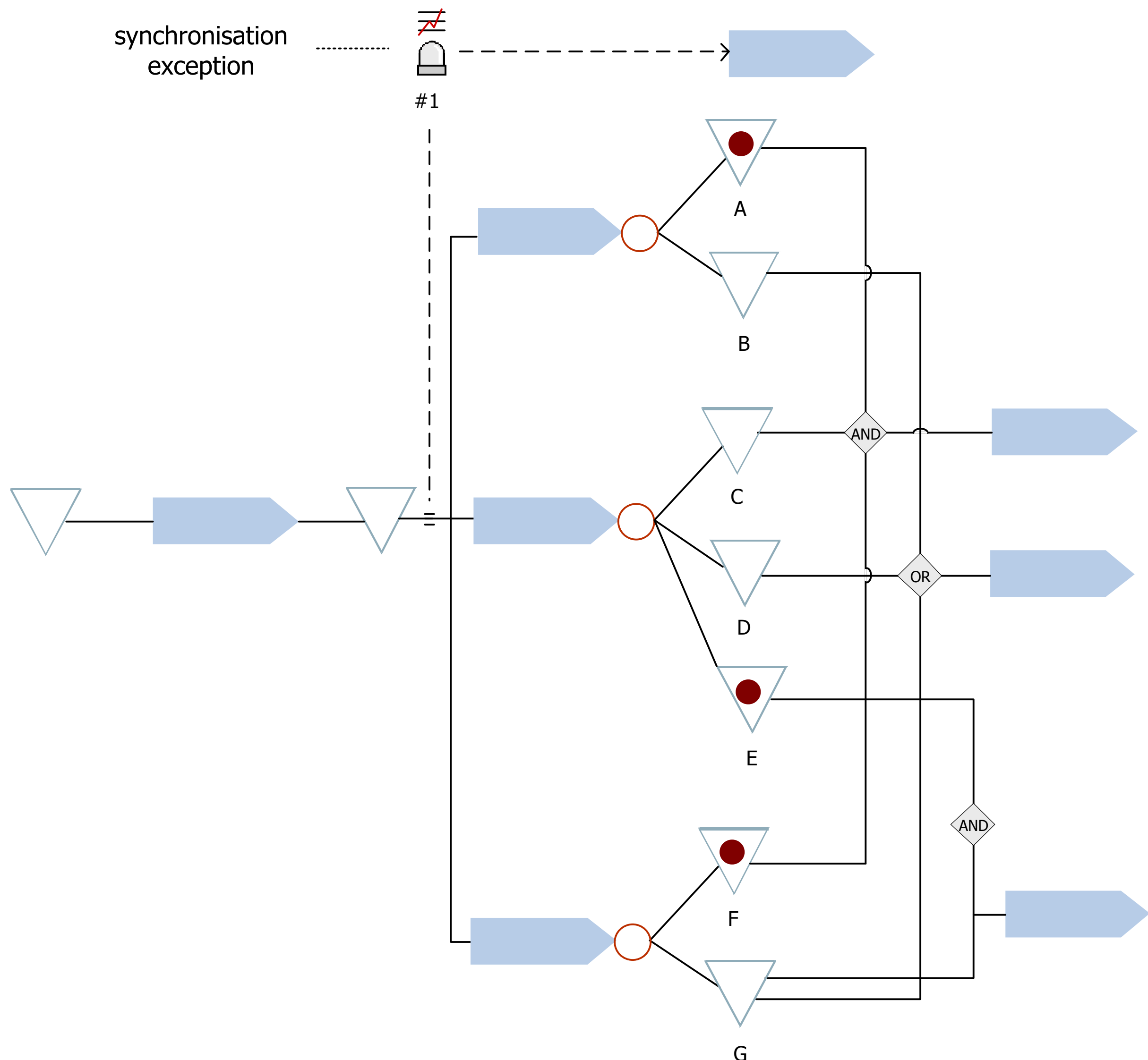


Figure 40: Use of a synchronisation exception

Abstract syntax: A synchronisation exception is associated to a set of concurrent trees (graphs). From a syntactical point of view it corresponds to a synchroniser. It can be used only, if the synchronisation of a set of concurrent trees (graphs) is not complete, i.e. if there are constellations of final events that are not accounted for. It triggers a path of execution that is alternative to any other path of execution resulting from explicit synchronisers. Like with any other exception, this exception handling path of execution will usually not be represented within the business process model, but separately in order not to increase a model's complexity too much. This requires assigning a unique name or identifier to a synchronisation exception. Unlike regular exceptions, the effect of a synchronisation exception cannot be specified as "resume".

Semantics: A synchronisation exception is a special kind of exception. It fires, if a set of concurrent paths of execution result in a pattern of events that is not covered by explicit synchronisers. If a syn-

chronisation exception fires, the corresponding concurrent paths of execution are synchronised and the business process continues with the corresponding exception handling process.

4.4 Exclusive Synchronisation

Splitting a business process in concurrent paths of execution will usually require division of labour and the subsequent coordination of the concurrent paths, i.e. their synchronisation. In the case of a disjunctional synchronisation, it is sufficient that one of the corresponding concurrent paths of execution terminates. It may happen, however, that one of the concurrent paths of execution is of outstanding importance, in the sense that its termination is sufficient for terminating the concurrent execution. In this case, a regular synchronisation is neither required nor appropriate. The following example illustrates such a case. It is a business process in a real estate company. The business process is started by a client who wants to sell a major property. The company's service includes the preparation of a professional sales brochure and an elaborate search for potential buyers. Since time is critical, both activities are started concurrently. The main purpose of the business process is selling the property. As soon as this purpose is accomplished, the business process will be sequentialised and all other concurrent processes are meaningless. In other words: If one particular final event occurs, the parallel execution is terminated. Note that this is different from a disjunctional synchronisation where a final event is associated with at least one further final event. There may be no feasible way to represent this kind of synchronisation by the use of conjunctional and/or disjunctional synchronisers. The example in Figure 41 shows a model of the outlined business process. It is assumed that there is only one possible outcome of the process "prepare brochure". The concept of an exclusive synchronisation allows for specifying one final event that immediately terminates the parallel execution. Note that more than one final event can be specified as an exclusive synchroniser. This is the case for the example in Figure 41: Not only finding a buyer but also the cancellation of the sales order will result in terminating the parallel execution. The use of disjunctional und conjunctional synchronisers is not sufficient for representing this kind of exclusive synchronisation. Note that the two alternative final events "buyer found" and "sales order withdrawn" must not be connected with the same synchroniser since they are not concurrent. It would require an additional negation operator (non "brochure created") for expressing that "buyer found" (or "sales order withdrawn") will terminate the parallel execution in any case. The business process model in Figure 42 illustrates this thought. It includes the explicit representation of the complementary event of "brochure created". In this case, the intended synchronisation can be expressed. However, it shows that the resulting synchronisation structure is likely to be confusing. Therefore, providing the concept of an exclusive synchroniser seems more appropriate then demanding for the representation of possibly irrelevant events and the construction of confusing synchronisation structures.

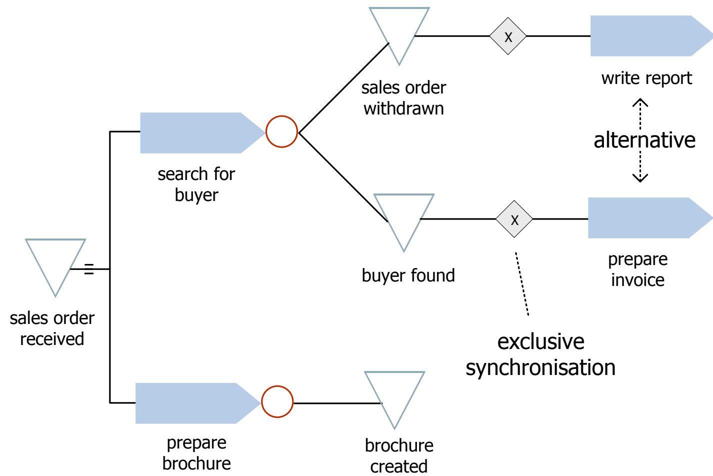


Figure 41: Example of an exclusive synchronisation (preliminary)

Abstract syntax: An exclusive synchronisation is associated with one or more alternative (not: concurrent) final events of a concurrent tree (graph) of execution. It is associated with the first process of a sequential path of execution. Within a set of concurrent final events more than one event may be associated with an exclusive synchroniser. If an exclusive synchronisation is applied, some final events of the concurrent trees of execution may not be associated with any synchroniser. This does not, however, contradict the rule that parallel paths of execution have to be synchronised (see above): As soon as an exclusive synchronisation fires, all remaining final events are regarded as synchronised.

Semantics: An exclusive synchronisation is a special case of synchronisation, where the occurrence of one specific final event causes the instant termination – and hence synchronisation – of a set of concurrent paths of execution. An exclusive synchroniser is always assigned to exactly one final event. For a set of concurrent paths of execution to be synchronised it is sufficient that all alternative final events of one concurrent path are associated with an exclusive synchronisation.

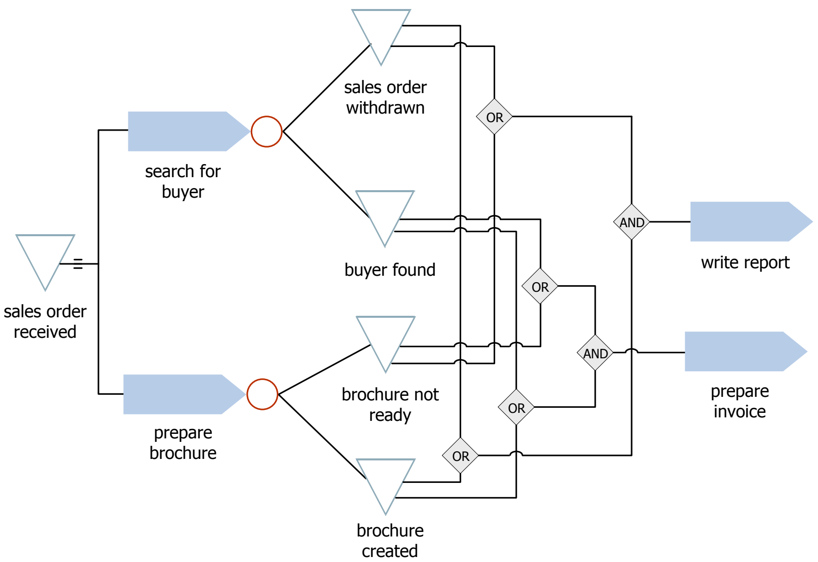


Figure 42: Alternative representation with additional final event (preliminary)

4.5 Relaxed Concurrency

Designing processes as concurrent allows for reducing the total execution time of a business process. Therefore it is a good choice, if execution time is critical and independent processes exist. There are cases, however, where the concurrent execution of processes makes sense, but the processes are not totally independent. Take, for instance, the example in Figure 41. Apparently, they are independent only to a limited degree: First, if the search for a buyer is successful, there is no need for a sales brochure anymore. Second, if the brochure is completed, the sales process is affected because it should make use of the brochure. While the second case is covered by the concept of an exclusive synchronisation, the first case requires a different approach.

In order to cope with this situation, it is required to notify the process "search for buyer" somehow, if the brochure is ready. This is accomplished by generating a special exception within the process "search for buyers". The exception is created by the event "brochure created". It serves to somehow notify the human actor or the machine that executes processes in parallel paths that may be affected. Note that this is kind of exception that we call a "concurrency exception" is different from a typical exception: First, it does not have to indicate a problematic or even irregular execution. Second, it must correspond to an event in a parallel path of execution.

Abstract syntax: A process within a concurrent tree (graph) can be assigned a special kind of exception, a concurrency exception. A concurrency exception is created by a corresponding event within a concurrent tree (graph) - including the final events. This correspondence is expressed by using the same designators both for the concurrency exception and the event.

Semantics: Two paths of execution are of relaxed concurrency, if it is possible that they are executed independently and if it is possible at the same time that an event produced by one path affects the execution of processes in the other path. Such an event creates a concurrency exception. A concurrency exception is meant to cause a meaningful adaptation of the process it is assigned to. The adaptation can be modelled by an exception handling process.

Figure 43 shows the slightly modified example in Figure 41 supplemented by a concurrency exception. To clarify that the corresponding paths of execution are of restricted independence, a special symbol for representing "relaxed concurrency" is used for the concrete syntax.

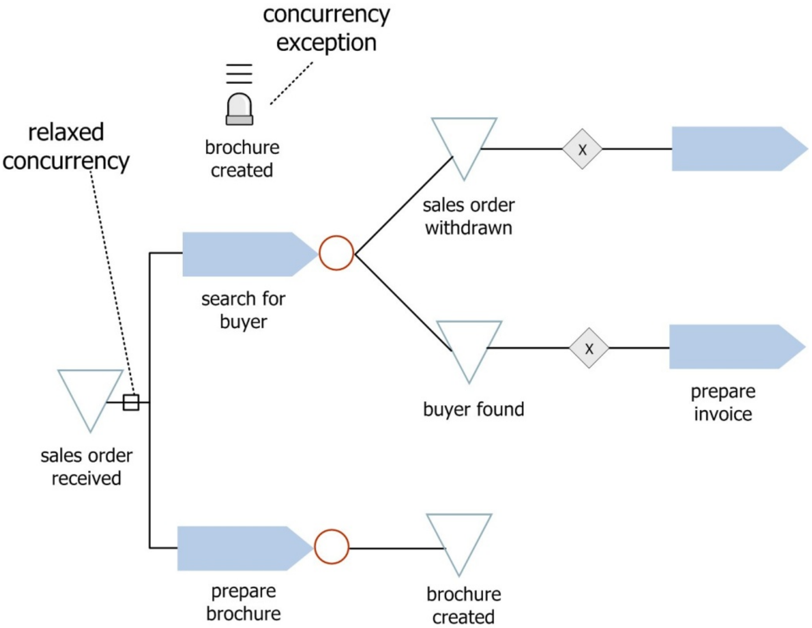


Figure 43: Relaxed concurrency, concurrency exception and exclusive synchronisation (preliminary)

4.6 Variable Number of Concurrent Instances

The concept of concurrency allows – as a special case – all concurrent trees (graphs) to be identical or – in other words – that instances of the same kind are executed in parallel. The number of these parallel instances may vary from one instance of the business process type to the other. The modelling concepts introduced so far do not allow for expressing a variable number of parallel paths (of the same kind). We doubt that this is a frequently required control structure. It should not be confused with multiple instances of one business process type. Russel et al. 2006) for instance, suggest a pattern called "Multiple Instances without Synchronisation". This is, however, more likely the case for independent instances of one business process types. The example they provide as an illustration confirms this interpretation.

To represent a variable number of concurrent instances, the path of execution that represents possible concurrent instances is marked by a special concurrency split. The regular symbol for concurrency splits is supplemented by a symbol that represents the number of executions. The number of executions can be left open to a decision made during run-time, it can be a constant or a reference to some variable (see illustration in Table 6).

numOfAgents	number of instances defined by reference to some object
n	number of instances to be determined during run-time
3	constant number of instances

Table 6: Symbols to represent the number of executions (preliminary)

The MEMO-OrgML does not cover all possible kinds of synchronisation for concurrent multiple instances. This is based on the assumption that only a few specific kinds of synchronisation are needed. Also, allowing for constructing more versatile synchronisation patterns would corrupt the simplicity that is intended with the conception of concurrent multiple instances.

Figure 44 shows an example of multiple concurrent instances, the number of which is determined during run-time and which are synchronised by a conjunctional synchroniser. It represents a business process in an advertising agency. The business process is triggered by a customer who wants the agency develop a new campaign. The final order depends on a presentation of one or more proposals. Depending on the volume of the order and the time pressure, it is decided how many alternatives are to be developed and how many teams should work on them in parallel. After all proposals prepared, there will be a presentation to the customer. The example in Figure 44 is supplemented by a corresponding model for the case of two concurrent instances where regular concurrency is used. While conjunctional synchronisation means that all parallel instances need to be terminated before sequentialising the process, it is also conceivable that a disjunctional synchronisation is used. It can be an option in the case of extreme time pressure: The concurrent paths would represent a race between different actors. As soon as the first path has finished, synchronisation would take place.

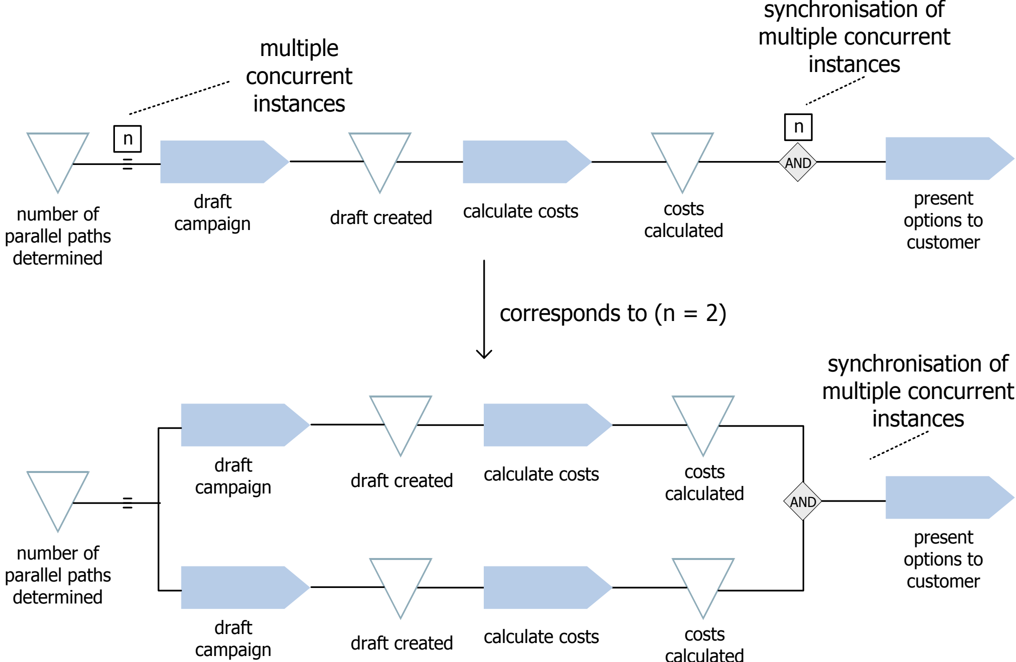


Figure 44: Example of multiple concurrent instances with conjunctional synchronisation (preliminary)

The parallel paths of execution in the above example produce one final event only. It is, however, possible that each instance results in more than one final event. The example in Figure 45, which is a variation of the example in Figure 44, shows such a case. Each of the alternative final events can be associated with a synchroniser. Since the final events are alternative, the synchronisers represent alternative results, too. This implies that they must be mutually exclusive, which is the case for the depicted combination of conjunctional and disjunctional synchronisers.

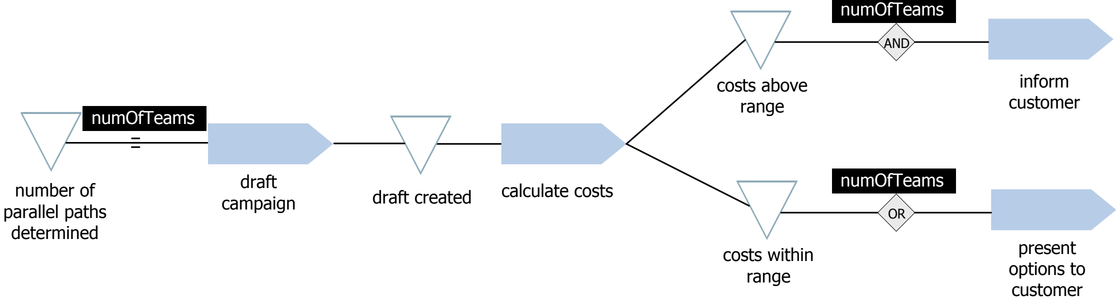


Figure 45: Multiple concurrent instances with alternative synchronisers (preliminary)

There is a further pattern of synchronisation that is characteristic for multiple concurrent instances. If, for instance, the business process in Figure 44 is supplemented by the constraint that the customer presentation has to take place at a certain date, it might be that not all of the concurrent instances will terminate in time. This can be expressed by supplementing the conjunctional synchroniser (this is not possible for disjunctional synchronisers) with a time symbol (either representing a time period or a point in time). It indicates that all instances that have terminated at the critical time are synchronised while the others are not accounted for (see Figure 46).

numOfTeams
=
number of parallel paths determined
draft campaign
draft created
calculate costs
costs calculated
numOfTeams
AND
present options to customer

Figure 46: Restricted conjunctional synchronisation (preliminary)

Abstract syntax: The abstract syntax of multiple concurrent instances corresponds to the syntax of concurrency. However, different from that, it depicts only one path of execution (a tree or a graph). The part of a business process model that is subject of multiple instantiations starts with special concurrency split and is terminated with a special synchroniser. The special concurrency split is associated with exactly one previous event. There are different kinds of concurrency splits for multiple concurrent instances: the number of instances can be predefined by a constant or by a variable or it can be left to a decision during run-time. Multiple concurrent instances can be synchronised only by a synchroniser that fits the corresponding concurrency split, e.g. a concurrency split that is characterised by a constant number of parallel instances allows for a synchroniser only that is also reserved for a constant number of parallel instances. The path of execution that represents multiple instances is terminated by one final event or by set of alternative final events. For semantic reasons, not more than two alternative final events are permitted. To synchronise the concurrent instances, each of the final events has to be associated with a synchroniser. The special synchronisers reserved for multiple concurrent instances must not be associated with further synchronisers.

Semantics: The concept of multiple concurrent instances corresponds to explicitly modelling n corresponding concurrent paths of execution. It can be interpreted as a shortcut, however, only in the case of a predefined, constant number of instances. Multiple concurrent instances cannot be explicitly represented, if the number of instances depends on a decision during run-time or on a variable. A conjunctional synchroniser means that all events of the kind, the synchroniser is associated with, have to fire before synchronisation. A disjunctional synchroniser means that synchronisation takes place as soon as one event of the kind, the synchroniser is associated with, fires. There is one further, special synchroniser. It represents what we call a restricted conjunctional synchronisation. It means that synchronisation accounts for those events of the kind, the synchroniser is associated with, that have fired until a certain point in time (or after a certain time period has passed). If multiple concurrent instances result in two synchronisers, one of them must be a disjunctional, the other a conjunctional synchroniser. Otherwise, it could be possible to produce inconsistent states: If both were conjunctional, a deadlock might occur. If both were disjunctional, they might not be mutually exclusive. It is not permitted to use a restricted conjunctional synchroniser together with a disjunctional or a further restricted conjunctional synchroniser. If it was used together with a disjunctional synchroniser, it would be possible that both synchronisers were not mutually exclusive. The same result could be produces if it was supplemented by a further restricted conjunctional synchroniser.

Note that the constraints relating to the synchronisers could also be regarded as part of the abstract syntax. They are subsumed under semantics because they basically reflect how synchronisers are to be

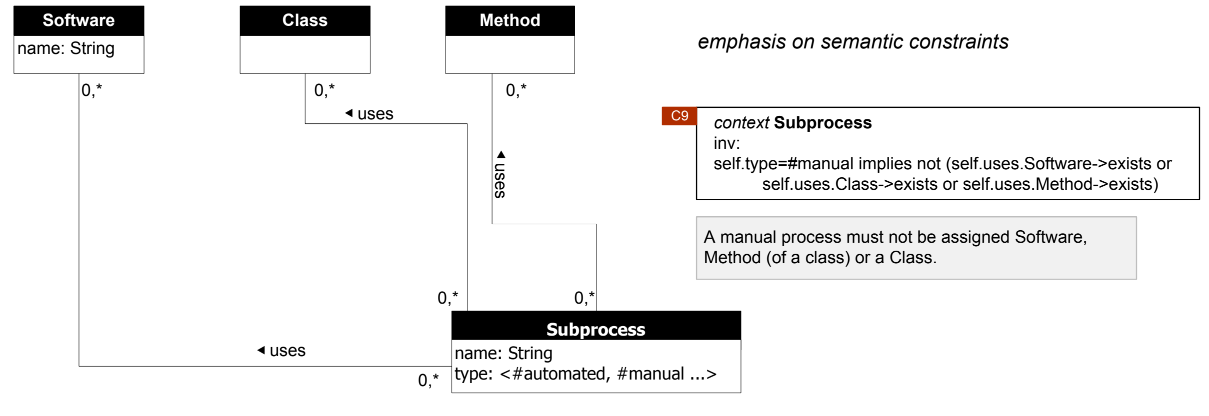
interpreted. With respect to the final language specification this does not exclude representing it at least in part through the abstract syntax

5 Language Specification

While the previous description of the language should be sufficient for most users, a more precise definition of the language's abstract syntax and semantics is required to enable the implementation of corresponding modelling tools. A meta model serves to specify the abstract syntax and semantics. In addition to that, the concrete syntax, i.e. the graphical notation, is presented as a catalogue of all notation elements.

5.1 Meta Model

Like with any conceptual model, the construction of a meta model should be based on thorough design decisions, which in turn should be based on a reflective design philosophy or specification style. Among other things, a design philosophy comprises a principal choice between putting more emphasis on syntactical or semantic constraints (see Frank 2010, pp. 47). Representing language concepts on a syntactical level comes with the advantage that there is less need for complex constraints. On the other hand, syntactical specifications cause problems for a corresponding modelling tool, if frequent changes to a model are to be expected. The following example illustrates the difference between the two specification styles.



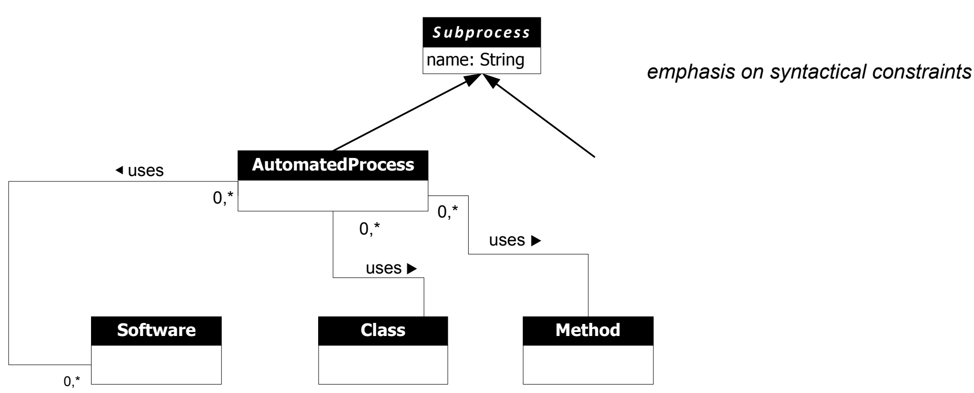


Figure 47: Difference between Specification Styles

A business process model will usually evolve during a process of communication with different stakeholders. As a consequence, it is very likely that it passes various stages of evolution which may be accompanied by substantial modifications, such as changing types of events or subprocesses. Therefore, the specification style chosen for the meta model puts emphasis on semantic constraints.

The documentation of the meta model comprises three parts: the meta model, constraints in OCL and a dictionary of all concepts. The specification of the meta model makes use of auxiliary types. They are presented in Table 7.

Specification	Comment
Money currency: String amount: Float	Money serves to specify financial amounts together with the respective currency.
Duration unit: **TimeUnit** dur: Float	Duration allows to define a time period using a selected unit of time.
ProcessType type: {#any, #auto, #semi_auto, #manual}	ProcessType is used to specify different types of subprocesses.
TimeEvent time: {#point, #interval}	TimeEvent allows for specifying whether an Event is created at a point in time or after a time interval has passed.
Notification type: {#humanNil, #humanSynch, #humanTradi, #humanElectronic, #softNil, #softPoll, #softPublish}	Notification serves the specification of different types of Events.
Change change:{#nil, #created, #modified, #deleted}	Change allows differentiating types of change that result in an Event.
Decision type: {#nil, #auto, #human, #clear}	Decision can be used to differentiate types of decision as they may be assigned to a Branching.
Cause cause: {#nil, #human, #failure, #timeOut}	Cause allows for specifying a type of cause for an Exception.
Effect effect: {#nil, #resume, #cancel}	Effect serves to specify whether an Exception results in terminating a BusinessProcess the respective Subprocess is part of or not.
Detection detect: {#nil, #auto, #likely, #chance}	Detection allows for specifying how an Exception is detected.
SynchType type: {#and, #or}	SynchType allows for specifying how parallel threads are to be synchronised.

Affirmation level: {#high, #medium, #low}	Affirmation serves expressing assessments.
Performance strengths: String weaknesses: String potential: String perfLevel: **Level**	Performance serves the differentiated specification of the performance related to a process.
Qualification description: String field: String level: String experience: {#no, #little, #satisfactory, #outstanding)	Qualification allows defining the actual or demanded qualification of a position, role, etc. In the case of business process modelling, it is used to describe qualification relevant for performing processes.

Table 7: Auxiliary Types

The meta model is specified in the MEMO Meta Modelling Language (Frank 2011a). The concepts of the meta model can be divided into two groups: Meta types are used to specify essential concepts for modelling business process types. They are instantiated into types and represented as such in corresponding business process models. They are also needed for executing respective process instances. Additional types, represented with the type name on a grey background, serve to represent modelling concepts that are not instantiated into types, but into instances that serve the purpose support the creation of syntactically valid models. They are not relevant for managing process instances. Note that AnyProcess is a special case: It serves to represent subprocess types that form an arbitrary or partially ordered sequence. Corresponding instances (subprocess types) are represented in a model. However, they would not be instantiated as such: The instantiation of an arbitrary sequence will always result in a particular sequence. Hence, the corresponding subprocesses would rather be instances of ControlFlowSubprocess. This may seem inconsistent at first. However, if one regards an instance of AnyProcess as an abstraction that represents a corresponding set of particular sequences, the instance-level of consist of an instantiation of one of these alternative sequences.

The representation of the meta model in Figure 48 includes references to constraints, which are specified in Table 8. Note that the meta model is not fully specified. Concepts like ExclusiveSynch and ConcurrencyException are underspecified (i.e. they lack additional constraints), because their semantics is still subject of further requirements analyses.

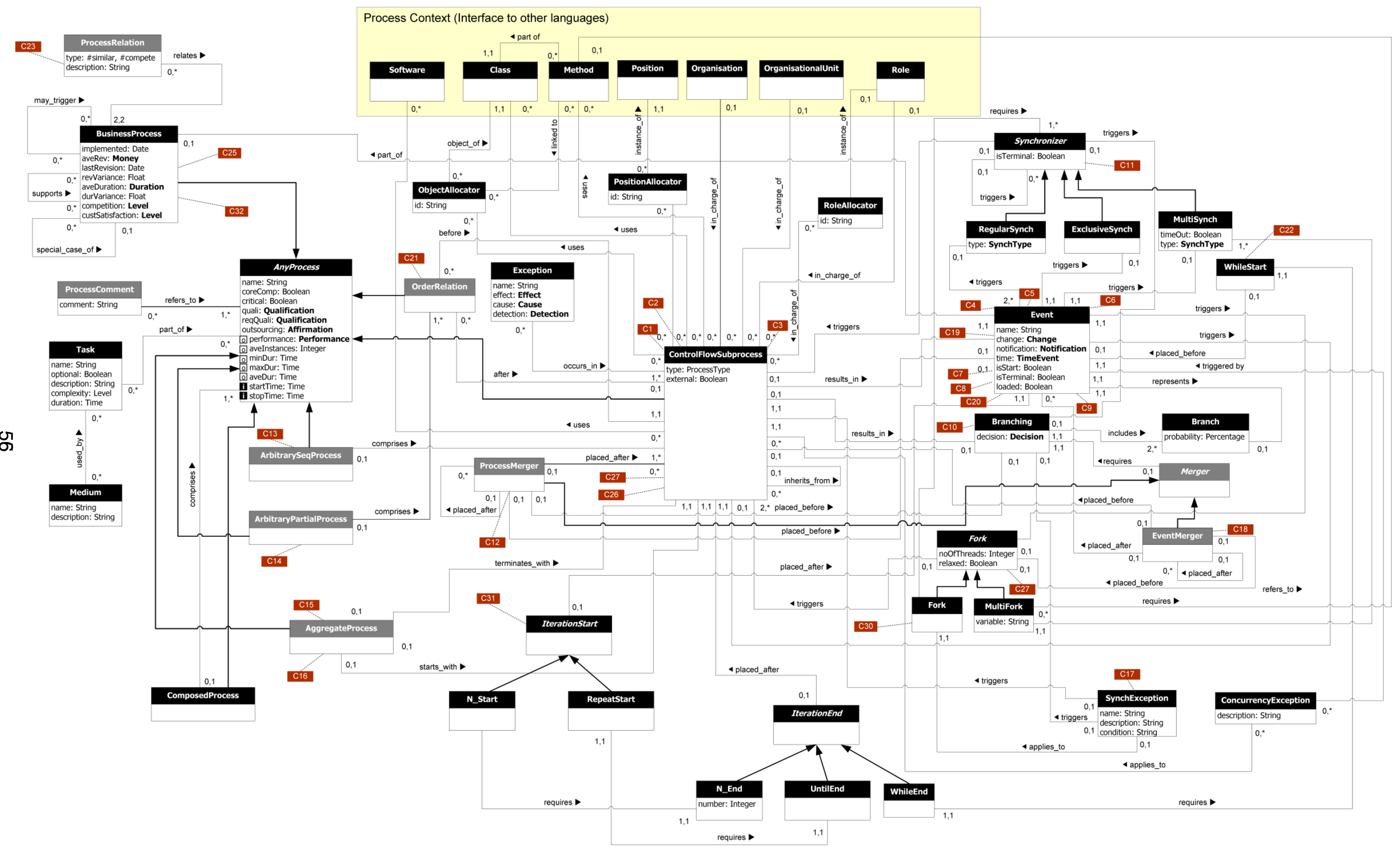
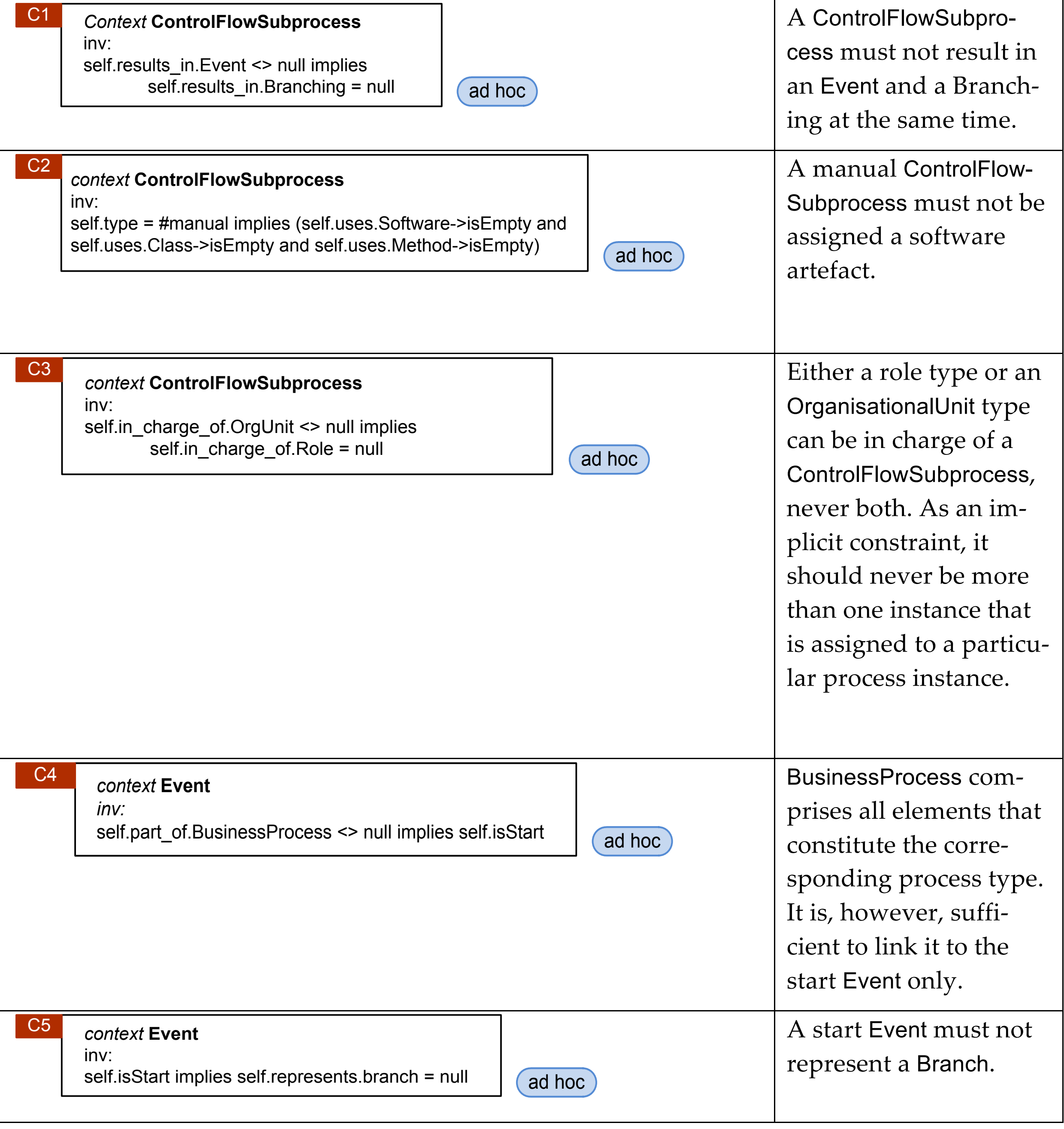


Figure 48: MEMO OrgML Meta Model - Focus on Processes

The constraints serve to promote model integrity. Some constraints can be applied on an ad hoc base, i.e. they allow for preventing particular edit operations immediately. For example: An event must not trigger a subprocess and a fork at the same time. Other constraints make sense only, if a process model is in a widely complete state. For instance: Every fork that starts a concurrent execution of threads requires at least one final synchroniser. However, it does not make sense to demand for the synchroniser as soon as the fork was created. Therefore, this group of constraints is supposed to be tested on demand only. In Table 8 ad-hoc constraints are marked as such.



	Constraint		Description
C1	*Context* **ControlFlowSubprocess** inv: `self.results_in.Event <> null implies self.results_in.Branching = null`	ad hoc	A ControlFlowSubprocess must not result in an Event and a Branching at the same time.
C2	*context* **ControlFlowSubprocess** inv: `self.type = #manual implies (self.uses.Software->isEmpty and self.uses.Class->isEmpty and self.uses.Method->isEmpty)`	ad hoc	A manual ControlFlowSubprocess must not be assigned a software artefact.
C3	*context* **ControlFlowSubprocess** inv: `self.in_charge_of.OrgUnit <> null implies self.in_charge_of.Role = null`	ad hoc	Either a role type or an OrganisationalUnit type can be in charge of a ControlFlowSubprocess, never both. As an implicit constraint, it should never be more than one instance that is assigned to a particular process instance.
C4	*context* **Event** *inv:* `self.part_of.BusinessProcess <> null implies self.isStart`	ad hoc	BusinessProcess comprises all elements that constitute the corresponding process type. It is, however, sufficient to link it to the start Event only.
C5	*context* **Event** inv: `self.isStart implies self.represents.branch = null`	ad hoc	A start Event must not represent a Branch.

<table>
<tr><td>C6
context **Event**
inv:
self.isTerminal implies (self.triggers.Software = null and
self.triggers.Subprocess = null and self.triggers.Fork = null and
self.triggers.Synchronizer = null)
ad hoc</td><td>A terminal Event must not trigger anything.</td></tr>
<tr><td>C7
context **Event**
inv:
self.triggers.ControlFlowSubprocess <> null implies not (self.triggers.For <>
null or self.triggers.RegularSynch <> null or self.triggers.ExclusiveSynch <>
null or self.triggers.MultiSynch <> null or self.placedBefore.EventMerger <>
null) and self.triggers.Fork <> null implies not
(self.triggers.ControlFlowSubprocess <> null or self.triggers.RegularSynch <>
null or self.triggers.ExclusiveSynch <> null or self.triggers.MultiSynch <> null
or self.placedBefore.EventMerger <> null) and self.triggers.RegularSynch <>
null implies not (self.triggers.ControlFlowSubprocess <> null or
self.triggers.Fork <> null or self.triggers.ExclusiveSynch <> null or
self.triggers.MultiSynch <> null or self.placedBefore.EventMerger <> null) and
self.triggers.MultiSynch <> null implies not
(self.triggers.ControlFlowSubprocess <> null or self.triggers.Fork <> null or
self.triggers.ExclusiveSynch <> null or self.triggers.RegularSynch <> null or
self.placedBefore.EventMerger <> null)
ad hoc</td><td>An Event must not trigger more than one subsequent element at a time.</td></tr>
<tr><td>C8
context **Event**
inv:
self.isTerminal implies not (self.isStart)
ad hoc</td><td>An Event cannot be a terminal and a start event at the same time.</td></tr>
<tr><td>C9
context **Event**
inv:
Event.allInstances()->forAll(a|if a.isStart then not
Event.allInstances()->exists(b|a<>b and b.isStart and
b.part_of = a.part_of) endif)
ad hoc</td><td>There must be no more than one start Event.</td></tr>
<tr><td>C10
context **Branching**
let result = 0.
inv:
self.includes.Branch->exists(a|a.percentage <> null) implies
(self.includes.Branch->iterate(a; | if a.percentage <> null then
result+a.percentage else result end)) and result = 100.</td><td>If a probability is assigned to a Branch, then the sum of all probabilities of all branches linked to one Branching must be 100.</td></tr>
<tr><td>C11
context **Synchronizer**
inv:
self.triggers.ControlFlowSubprocess <> null
implies not (self.triggers.Branching <> null or
self.triggers.Synchronizer->isEmpty())
self.triggers.Branching <> null
implies not (self.triggers.ControlFlowSubprocess
<> null or self.triggers.Synchronizer->isEmpty())
ad hoc</td><td>A Synchroniser must not trigger more than one subsequent elements at a time.</td></tr>
<tr><td>C12
context **ProcessMerger**
inv:
self.placed_before.Event <> null implies not
(self.placed_before.Branching <> null)
ad hoc</td><td>A ProcessMerger must not be placed before an Event and a Branching at the same time.</td></tr>
</table>

C13 *context* **ArbitrarySeqProcess** *inv:* self.type = null. ad hoc	Since a process type does not make sense for these processes (except for the case that all processes they comprise are of the same type), these constraints excludes assigning a type – and at the same time enables to take advantage of specialising them from ControlFlowSubprocess. This is a pragmatic – not an elegant - solution.
C14 *context* **ArbitraryPartialProcess** *inv:* self.type = null. ad hoc	
C15 *context* **AggregateProcess** *inv:* self.type = null. ad hoc	
C16 *context* **AggregateProcess** *inv:* self.startsWith <> self.terminatesWith. ad hoc	An aggregate process must not be restricted to one subprocess.
C17 *context* **SynchException** inv: self.triggers.ConrolFlowSubprocess <> null implies (self.triggers.Branching = null) ad hoc	A SynchException must not trigger a ControlFlowSubprocess and a Branching at the same time.
C18 *context* **EventMerger** *inv:* self.placed_before.ControlFlowSubprocess <> null implies (self.placed_before.Fork = null) ad hoc	An EventMerger must not be placed before a ControlFlowSubprocess and a Fork at the same time.
C19 *context* **Event** inv: Event.allInstances()->exists (e | e.isTerminal = true)	There has to be at least one terminal event type within a business process model. Note that this is a weak constraint, since it does not account for alternative paths of execution each of which may results in a separate terminal event.

C20 *context* **Event** inv: self.loaded implies self.triggers.Fork <> null. ad hoc	An event type can be marked as loaded only, if it is placed before a fork.
C21 *context* **OrderRelation** *inv:* self.before <> null or self.after <> null.	There must be a subprocess either before or after any subprocess within an ArbitraryPartialProcess.
C22 *context* **WhileStart** *inv:* self.placedBefore.isStart = false. ad hoc	A WhileStart must not be placed before a start Event.
C23 *context* **ProcessRelation** def: let br: self.relates.asOrderedSet() inv: br.at(1) <> br.at(2) ad hoc	Two BusinessProcess types that are related must be different.
C25 *context* **BusinessProcess** *def*: let allSuperTypes: collect (me | me = me.special_case_of) *inv*: (self.allSuperTypes-> includes self) = false ad hoc	Specialisation associations between BusinessProcess must not be cyclic.
C26 *context* **ControlFlowSubprocess** inv: self.results_in.Event <> self.triggered_by.Event ad hoc	The event type that triggers a subprocess type must be different from the event type that results from a subprocess type.
C27 *context* **ControlFlowSubprocess** *def*: let allSuperTypes: collect (me | me = me.inherits_from) *inv*: (self.allSuperTypes-> includes self) = false ad hoc	Inheritance associations between subprocess types must not be cyclic.
C28 *context* **EventMerger** *Inv:* self.eventMerger->size() + self.event->size() > 1. ad hoc	An event merger needs to link at least two elements – which may be event types and/or other event mergers.
C29 *context* **ProcessMerger** *Inv:* self.controlFlowSubprocess->size() + self.processMerger->size() > 1.	A process merger needs to link at least two elements – which

ad hoc	may be subprocess types and/or other process mergers.

C30

```
context Fork
def: successors (in: Element) : Sequence(Entity) =
            if elt.oclIsKindOf(ControlFlowSubprocess) then
        if elt.results_in.Event <> null then
            Set{elt.results_in.Event}
        elseif not elt.results_in.Branching->isEmpty()
            elt.results_in.Branching
        elseif ProcessMerger.allInstances()->exists(pm|pm.placed_after=elt) then
            ProcessMerger.allInstances()->select(pm|pm.placed_after=elt)
        endif
            elseif elt.oclIsKindOf(Event) then
        if elt.triggers.ControlFlowSubprocess <> null then
            Set{elt.triggers.ControlFlowSubprocess}
        elseif elt.triggers.RegularySynch then
            Set{elt.triggers.RegularySynch}
        elseif elt.triggers.ExclusiveSynch then
            Set{elt.triggers.ExclusiveSynch}
        elseif elt.triggers.MultiSynch then
            Set{elt.triggers.MultiSynch}
        elseif not elt.triggers.Fork->isEmpty() then
            elt.triggers.Fork
        elseif EventMerger.allInstances()->exists(em|em.placed_after = elt) then
            EventMerger.allInstances()->select(em|em.placed_after = elt)
        endif
    elseif elt.oclIsKindOf(Synchronizer) then
        elt.triggers.ControlFlowSubProcess
    elseif elt.oclIsKindOf(Fork) then
        elt.triggers.ControlFlowSubProcess
    elseif elt.oclIsKindOf(Branching) then
        elt.including->iterate(branch res = Set{}|
            Set{}.including(Event.allInstances()->select(a|a.represents = branch)->iterate(a result = null| a))
        )
    elseif elt.oclIsKindOf(EventMerger) then
        if elt.placed_before.ControlFlowSubProcess <> null then
            Seq{elt.placed_before.ControlFlowSubProcess }
                    elseif elt.placed_before.Fork <> null then
            Seq{elt.placed_before.Fork}
        end
            elseif elt.oclIsKindOf(ProcessMerger) then
                if elt.placed_before.Branching <> null then
                    Seq{elt.placed_before.Branching}
                elseif elt.placed_before.Event <> null then
                    Seq{elt.placed_before.Event}
                endif
            endif
endif
def: findRadical(start:Element,findWhat:Set(),forks:Set(),synchronizers:Set(),visited:Set()):Boolean =
    if visited.includes(start) then
        /* If a node has been visited more than once  --> terminate iteration */
                    false
    elseif findWhat.includes(start) then
        /* when arrived at the synchronizer, all intermediate forks and synchronizers */
                    /* must have been visited already */
                    forks->forAll(fork|requires->exists(requirement|synchronizers.includes(requirement)))
                    and
                    /* for each synchronizer a corresponding fork must have been visited */
                    synchronizer->forAll(sync | forks->exists(fork| fork.requires.includes(sync)))
            else
        if successors(start)->isEmpty() then
                            /* if the required synchronizers are still missing and no successors */
                            /* are available, terminate */
                            false
        else
        /* otherwise search beginning with all successors of the current element  */
        /* for synchronizers. Add current element to already visited elements */
                /* collect all succeeding elements that are kind of Fork and all succeeding elements */
                /* that are kind of Synchronizer */
                            successors(start)->forAll(suc|
                findRadical(suc, findWhat,
                                    forks+successors(start)->select(fork|fork.oclIsKindOf(Fork)),
                                    synchronizers+successors(start)-
                                    >select(fork|fork.oclIsKindOf(Synchronizer)),
                                    visited.including(start)))
        endif
    endif
/* try finding all synchronizers that correspond to a fork using a breadth-first search */
/* that includes all intermediate forks */
inv: findRadical(self,self.requires,Set{},Set{},Set{})
```

All concurrent paths of a parallel execution need to be synchronised appropriately. For an illustration see Figure 13.

Constraint	Description
C31	Each path of execution that starts with an Iteration Start, must end in a corresponding iteration end. This applies to intermediate iterations, too.

```
context IterationStart
def: successors (in: Element) : Sequence(Entity) =
        if elt.oclIsKindOf(ControlFlowSubprocess) then
      if elt.results_in.Event <> null then
         Set{elt.results_in.Event}
      elseif not elt.results_in.Branching->isEmpty()
         elt.results_in.Branching
      elseif ProcessMerger.allInstances()->exists(pm|pm.placed_after=elt) then
         ProcessMerger.allInstances()->select(pm|pm.placed_after=elt)
      endif
        elseif elt.oclIsKindOf(Event) then
      if elt.triggers.ControlFlowSubprocess <> null then
         Set{elt.triggers.ControlFlowSubprocess}
      elseif elt.triggers.RegularySynch then
         Set{elt.triggers.RegularySynch}
      elseif elt.triggers.ExclusiveSynch then
         Set{elt.triggers.ExclusiveSynch}
      elseif elt.triggers.MultiSynch then
         Set{elt.triggers.MultiSynch}
      elseif not elt.triggers.Fork->isEmpty() then
         elt.triggers.Fork
      elseif EventMerger.allInstances()->exists(em|em.placed_after = elt) then
         EventMerger.allInstances()->select(em|em.placed_after = elt)
      endif
   elseif elt.oclIsKindOf(Synchronizer) then
      elt.triggers.ControlFlowSubProcess
   elseif elt.oclIsKindOf(Fork) then
      elt.triggers.ControlFlowSubProcess
   elseif elt.oclIsKindOf(Branching) then
      elt.including->iterate(branch res = Set{}|
         Set{}.including(Event.allInstances()->select(a|a.represents = branch)->iterate(a result
= null| a))
      )
   elseif elt.oclIsKindOf(EventMerger) then
      if elt.placed_before.ControlFlowSubProcess <> null then
         Seq{elt.placed_before.ControlFlowSubProcess }
               elseif elt.placed_before.Fork <> null then
         Seq{elt.placed_before.Fork}
      end
        elseif elt.oclIsKindOf(ProcessMerger) then
           if elt.placed_before.Branching <> null then
              Seq{elt.placed_before.Branching}
           elseif elt.placed_before.Event <> null then
              Seq{elt.placed_before.Event}
           endif
        endif
endif
def:
findRadical(start:Element,findWhat:Element,iterationStarts:Set(),iterationEnds:Set(),visited:Se
t()):Boolean =
   /* starts with the current element – searching all paths for the required element. Required
element is searched for with findWhat, which represents a breadth-first search */

        if visited.includes(start) then
      /* If an element had been visited -> iteration & termination */
               false
   elseif start = findWhat then
      /* If iteration end was reached, check whether all visited iteration starts could be closed
*/
               /* within the analysed paths */
               iterationStarts->forAll(is | iterationEnds.includes(is.requires))
      and
      /* Iteration ends have to be started respectively closed within the respective paths */
               iterationEnds->forAll(ie | iterationStarts->exists(is|is.requires))
        else
      if successors(start)->isEmpty() then
         /* if no successors exist, terminate (fail) */
                     false
      else
         /* recursive call including all successors of the current element */
         /* All iteration starts and iteration ends need to be memorized */
         /* and each current element is added to those already visited */
                     successors(start)->forAll(suc|
         findRadical(suc, findWhat,
            iterationStarts+IterationStart.allInstances()->select(is | successors(start)
                                    ->exists(a|is.placedBefore = a)),
            iterationEnds+IterationEnd.allInstances()->select(ie | successors(start)
                                    ->exists(a|ie.placedBefore = a),
            visited.including(start)))
      endif
   endif
inv: findRadical(self.placed_after,self.requires.placed_after,Set{},Set{},Set{})
```

C32

```
context BusinessProcess
/* detect successor of an element, returns a sequence */
def: successors (in: Element) : Sequence(Entity) =
            if elt.oclIsKindOf(ControlFlowSubprocess) then
        if elt.results_in.Event <> null then
            Set{elt.results_in.Event}
        elseif not elt.results_in.Branching->isEmpty()
            elt.results_in.Branching
        elseif ProcessMerger.allInstances()->exists(pm|pm.placed_after=elt) then
            ProcessMerger.allInstances()->select(pm|pm.placed_after=elt)
        endif
            elseif elt.oclIsKindOf(Event) then
        if elt.triggers.ControlFlowSubprocess <> null then
            Set{elt.triggers.ControlFlowSubprocess}
        elseif elt.triggers.RegularySynch then
            Set{elt.triggers.RegularySynch}
        elseif elt.triggers.ExclusiveSynch then
            Set{elt.triggers.ExclusiveSynch}
        elseif elt.triggers.MultiSynch then
            Set{elt.triggers.MultiSynch}
        elseif not elt.triggers.Fork->isEmpty() then
            elt.triggers.Fork
        elseif EventMerger.allInstances()->exists(em|em.placed_after = elt) then
            EventMerger.allInstances()->select(em|em.placed_after = elt)
        endif
    elseif elt.oclIsKindOf(Synchronizer) then
        elt.triggers.ControlFlowSubProcess
    elseif elt.oclIsKindOf(Fork) then
        elt.triggers.ControlFlowSubProcess
    elseif elt.oclIsKindOf(Branching) then
        elt.including->iterate(branch res = Set{}|
            Set{}.including(Event.allInstances()->select(a|a.represents = branch)->iterate(a result = null| a))
        )
    elseif elt.oclIsKindOf(EventMerger) then
        if elt.placed_before.ControlFlowSubProcess <> null then
            Seq{elt.placed_before.ControlFlowSubProcess }
                    elseif elt.placed_before.Fork <> null then
            Seq{elt.placed_before.Fork}
        end
            elseif elt.oclIsKindOf(ProcessMerger) then
                if elt.placed_before.Branching <> null then
                    Seq{elt.placed_before.Branching}
                elseif elt.placed_before.Event <> null then
                    Seq{elt.placed_before.Event}
                endif
            endif
endif
def: findRadical(start:Element,visited:Set()):Boolean =
    if visited.includes(start) then
        /* Falls Zirkulation -> Abbruch */
                    false
    elseif start.isKindOf(Event) and start.isTerminal then
        /* Falls terminierendes Event -> positiv */
                    true
    else
        if successors(start)->isEmpty() then
            /* if there is no successing element, terminate */
                            false
        else
            /* repeat procedure with successor of current element
            /* and add current element to already visited elements */
                            successors(start)->forAll(suc|findRadical(suc,visited.including(start)))
        endif
    endif
  inv: findRadical(Event.allInstances()->select(a|a.isStart and a.partOf(self)),Set{})
```

Each business process has to terminate in one or more end events.

Table 8: Constraints

5.2 Language Concepts

The specification of language concepts in the meta model and supplementing constraints is demanding to read and understand. The following comments on selected language concepts are supposed to foster a better understanding of the specification. The first of the multiplicities that are shown for each association is linked to the focussed meta type – presented on top of the two.

AnyProcess	This is an abstract meta type that serves specifying core features of processes.	
Subtypes		
ControlFlowSubprocess, BusinessProcess		
Attributes on type level		
name	String	Allows for assigning a type name. Note that the name should be unique within the context of a business process type. Nevertheless, there is no respective constraint, because there may be cases where it seems appropriate to assign more than one (sub) process the same name.
coreComp	Boolean	If coreComp is set to true, this indicates that performing the corresponding process type represents a core competence.
critical	Boolean	Serves to specify whether a process type is of critical competitive relevance. If both, coreComp and critical are set to true, the corresponding process can be regarded as a core process.
quali	Qualification	Serves to specify the actual qualification that is – on average – available for performing processes of this type.
reqQuali	Qualification	Serves to specify the qualification that is required for performing processes of this type on a satisfactory level.
outsourcing	Affirmation	Allows for expressing to what degree the corresponding process type is a candidate for being outsourced.
Attributes with reference to instance level		
performance <<obtainable>>	Performance	Serves the description of the performance related to a process. A corresponding value may be obtained from an external system that monitors the performance of instances.
aveInstances <<obtainable>>	Integer	Stores the average number of instances within a certain time period, e.g. the months of a year. The value can be obtained from an external system such as a Workflow Management System.
minDur <<obtainable>>	Time	Serves the description of the minimum duration of processes of this type. It can be either defined or obtained from an external system.
maxDur <<obtainable>>	Time	Serves the description of the maximum duration of processes of this type. It can be either defined or obtained from an external system.
aveDur <<obtainable>>	Time	Serves the description of the average duration of processes of this type within a certain time period. It can be either defined or obtained from an external system.
startTime <<intrinsic>>	Time	This attribute is intrinsic, i.e. it applies to instances of corresponding types only and serves to store the start time of a particular instance.
stopTime <<intrinsic>>	Time	This attribute is intrinsic, too, and serves to store the termination time of a particular instance.
Associations		
part_of	1,*	with ComposedProcess. This association corresponds to the Composite Pattern. Its purpose is to allow for process decomposition diagrams. Note

comprises	0,1	that this abstraction comes with a disadvantage: On the highest level, a composite process is an entire business process type, hence, should be represented by an instance of BusinessProcess. However, the language specification does not allow for expressing that. Instead, an instance of ComposedProcess would be used to represent business process type.
described_by refers	1,* 0,*	with ProcessComment. One or more comments can be assigned to any process type.
includes is_part_of	0,* 0,*	A process may include various tasks. In case the order of performing tasks is relevant, tasks should be modelled as subprocess types.

Table 9: Comments on *AnyProcess*

The two following concepts are specialisations of the previous one.

BusinessProcess	BusinessProcess is instantiated into business process types. The meaning of a business process type is defined through the control flow it refers to by pointing to the start Event. Business process types can be used within business process maps, which requires various kinds of relationships between business process types. Furthermore, they serve for representing features that apply to the entire type rather than to its components. Features inherited from the supertype are not described again.	
Example Instances		
"Claims Handling", "Order Management"		
Supertype		
AnyProcess		
Attributes on type level		
implemented	Date	Serves to specify the data when a business process type was first implemented.
aveRev	Money	Serves to store the average revenues of corresponding process instances within a certain time period.
lastRevision	Date	Serves to specify the data when the corresponding business process type was last revised.
revVariance	Float	Serves to store the variance of revenues of corresponding process instances within a certain time period.
aveDuration	Duration	Serves to store the average duration of corresponding process instances within a certain time period.
durVariance	Float	Serves to store the variance of durations of corresponding process instances within a certain time period.
competition	Level	Serves to specify the level of competition the corresponding business process type has to deal with.
custSatisfaction	Level	Allows for specifying the level of customer satisfaction.

Associations		
related_by relates	2,2 0,*	with ProcessRelation . It serves to define undirected associations like similarity or competition between business process types. There are always two different business process types associated with zero or many instances of ProcessRelation.
may_trigger triggered_by	0,* 0,*	with `self.` Allows to express that instances of zero or more business process types may trigger instances of one of more business process types. While this will usually be a different business process type, a corresponding constraint is not specified, because it is conceivable that an instance of a business process type triggers a further instance of the same type.
supports supported_by	0,* 0,*	with `self.` Allows to express that instances of zero or more business process types support instances of zero or more business process types. While this will usually be a different business process type, a corresponding constraint is not specified, because it is conceivable that an instance of a business process type supports a further instance of the same type.
special_case_of specialized_to	0,* 0,1	with `self.` This association serves to express that zero or more business process types can be specialised from zero or one further business process type. Note that there is no formal semantics specified for this association – except that it must not be cyclic. Therefore it serves only to express an assessment of a modeller, which may be utilised for maintenance purposes.

Constraints	
C25	There are serious reasons why a conception of process specialisation that satisfies the substitutability constraint is not feasible (see, e.g. Frank 2012). Therefore, the only constraint that is specified serves to prevent cyclic association.

Table 10: Comments on BusinessProcess

The following concept, ControlFlowSubprocess, represents subprocess types. The MEMO OrgML allows for differentiation various types of subprocesses, such as manual, automated etc. There are two design alternatives to represent this variety of possible types. On the one hand, one could explicitly specify corresponding meta types, e.g. as specialisations of a common meta type. This approach would have the advantage that specific constraints, e.g. that a manual subprocess type must not be associated with Software, could be guaranteed on a syntactical level. It comes, however, with the disadvantage that changing the type of a subprocess during the lifetime of a model will require changing the type/class of the respective element during run-time of a corresponding tool. Therefore, the specification is based on the second option, where the type of a subprocess is defined using an attribute. As a consequence, a corresponding constraint can be checked only by checking the state of the instance of ControlFlowSubprocess. One does not have to be enthusiastic about such a specification, but seems to be the lesser of two evils.

ControlFlowSubprocess	ControlFlowSubprocess is instantiated into subprocess types of a business process type. Each subprocess type can be represent a certain type of process, e.g. manual, automated etc. Features inherited from the supertype are not described again.	
Supertype		
AnyProcess		
Example Instances		
"Check Credibility", "Register Order", "Prepare Invoice"		
Attributes on type level		
type	ProcessType	Allows for specifying the particular type of the subprocess type.
external	Boolean	Serves to specify that instances of this subprocess type are executed within an external organisation.
Associations		
placed_before placed_after	2,* 0,1	with ProcessMerger. Two or subprocess types that are part of alternative paths of execution can be merged by using a ProcessMerger.
terminates_with terminal_process_of	1,1 0,1	with AggregateProcess.
generates occurs_in	1,* 0,*	with Exception. An exception type may occur in one or many subprocess types. A subprocess type can generate zero to many exception types.
uses used_by	0,* 0,*	with ObjectAllocator. ObjectAllocator serves to express that a specific object is used by a subprocess type. Note that this is not a particular instance. Instead, ObjectAllocator allows identifying/distinguishing potential instances of a class throughout a business process type. Each ObjectAllocator is associated with exactly one class represents an instance of. It can also be associated with a method. Similar associations include those with PositionAllocator and RoleAllocator – however, the meaning of these associations is different.
managed_by in_charge_of	0,* 0,1	with OrganisationalUnit. Serves to assign an organisational unit that is in charge of the subprocess type. Similar associations include those with Class, Method, Position, Role, and Organisation – however, the meaning is different in the case of Class and Method.
placed_before placed_after	1,1 0,1	with IterationEnd. A subprocess type is placed before each instance of IterationEnd. Zero to one instances of IterationEnd may be placed after an IterationEnd.
triggered_by triggers	0,1 0,1	with Event. A subprocess type is triggers zero to one Event types and is triggered by zero to one Event types.
triggered_by triggers	2,* 0,1	with Fork. An instance of Fork triggers at least two instances of ControlFlowSubprocess (i.e. a subprocess type).
placed_after placed_before	0,1	with EventMerger. An instance of ControlFlowSubprocess can be placed after or before zero to one mergers.

	0,1	
triggered_by triggers	0,1 0,1	with SynchException. An instance of SynchException trigger exactly one subprocess type.
results_in produced_by	1,1 0,1	with Branching. An instance of Branching always results from an instance of a subprocess type.
results_in produced_by	0,1 0,1	with Event. A subprocess type results in zero or one event types.
triggered_by triggers	0,1 0,1	with Synchronizer. The synchronisation of concurrent paths of execution triggers either a subprocess type or a branching.
before after	0,1 0,*	with OrderRelation. This association serves to specify the temporal order of execution of two subprocess types within a partially ordered arbitrary process. A partially ordered process type (instance of ArbitraryPartialProcess) represents a set of subprocess types that are executed sequentially without a predefined temporal order.
part_of comprises	2,* 1,1	with ArbitrarySeqProcess. Two or more subprocess types can be assigned to an arbitrary sequential process. A arbitrary sequential process type (instance of ArbitrarySeqProcess) represents a set of subprocess types that are executed sequentially without a predefined temporal order, however, with a few constraints that define partial execution orders.
is_terminal_of terminates_with	1,1 0,1	with AggregateProcess. A subprocess type can mark the final subprocess of an aggregate process. Note that an aggregate process must not start with a further aggregate process.
is_start_of starts_with	1,1 0,1	with AggregateProcess. A subprocess type can mark the starting subprocess of an aggregate process.
inherits_from inherits_to	0,* 0,1	with self. A specialisation of process types that would satisfy the substitutability constraint is no possible. The "inherits_from" association represents a pragmatic compromise. It is restricted to inheriting tasks and methods (of classes) from a super process type. Inherited tasks should not be overridden (for methods this is not possible anyway, because there definition is out of the scope of the OrgML). Inheritance relationships must not be cyclic. Note that this is a preliminary concept to foster some degree of reuse. Therefore, further constraints, e.g. to prevent a manual process to inherit from an automated or the other way around are not specified yet.

Constraints	
C1	A ControlFlowSubprocess must not result in an Event and a Branching at the same time.
C2	If the process type of an instance of ControlFlowSubprocess is set to #manual, it must not be assigned a software artefact.
C3	Either a role type or an OrgansationalUnit type can be in charge of a ControlFlowSubprocess, never both. As an implicit constraint, it should never be more than one instance that is assigned to a particular process instance.

C26	The event type that triggers a subprocess type must be different from the event type that results from a subprocess type.
C27	Inheritance associations between subprocess types must not be cyclic.

Table 11: Comments on ControlFlowSubprocess

The meta type Event is of outstanding relevance for the specification of possible control flow structures. It is associated to many other concepts and supplemented with various constraints.

Event	Event is instantiated into event types that form an essential part of the control flow. Each event type can represent a certain event category or type. Event categories can be combined.	
Example Instances		
"Order received", "Amount not sufficient", "Budget exceeded"		
Attributes on type level		
name	String	Serves assigning a name to an event type. The name of an event type will usually, but must not be unique within the scope of a business process type.
change	Change	Serves specifying the kind of change that causes the event, e.g. creation, deletion, modification of an object.
notification	Notification	Serves specifying how the responsible human or the software system that manages a process get notified of a corresponding event.
time	TimeEvent	Serves specifying whether the event type is related to temporal aspects, i.e. to a point in time or to a time period.
isStart	Boolean	Allows marking an event type as representing start events.
isTerminal	Boolean	Allows marking an event type as representing terminal events.
loaded	Boolean	Allows marking an event type as loaded, i.e. as ambiguous with respect to the related notification mechanisms. It is applicable before concurrent paths of execution only: An event may lead to different types of notification in the various paths of execution. It cannot be combined with other kinds of events.
Associations		
triggered_by triggers	1,1 0,1	with BusinessProcess. On event type only can "trigger" a corresponding business process type: the one that represents the start event.
triggers triggered_by	2,* 0,1	with RegularSynch. A regular synchroniser within a parallel paths of execution is triggered by two or more event types.
triggers triggered_by	1,* 0,*	with ExclusiveSynch. An exclusive synchroniser is triggered by exactly one event type. Since this event type represents a number of event types, a synchronisation type (OR, AND) has to be specified (see example in Figure 45)

triggers triggered_by	1,1 0,1	with MultiSynch. A multiple synchroniser is used in cases where the concurrent paths of execution are not modelled explicitly. It is attached to a final event type that represents a possible outcome of
triggers triggered_by	1,1 0,1	with Fork. A parallel execution is always triggered by exactly one event.
triggers triggered_by	0,1 0,1	with ControlFlowSubprocess. A subprocess type is triggered by zero or one event. If it is not triggered by an event, it is triggered by a fork.
placed_after placed_before	0,1 1,1	with WhileStart. The beginning of a "while" iteration is placed before exactly one event type.
represents represented_by	1,1 0,1	with Branch. An instance of Event, i.e. an event type, can be associated with an instance of Branch to represent a branching decision.
placed_before placed_after	2,* 0,1	with EventMerger. Two or more event types can be connected to an event merger, if they represent alternative paths of execution.
placed_before placed_after	1,1 0,1	with IterationStart. IterationStart is an abstract concept that is specialised into N_Start and RepeatStart, which represent the beginning of a "for" loop and a "repeat until" loop respectively. Each of them is placed after exactly one event type.
placed_after placed_before	0,1 0,1	with ProcessMerger. After two subprocess types that belong to alternative paths of execution have been merged using an instance of ProcessMerger, this will always result in a subsequent event type or in a branching.
produced_by results_in	0,1 0,1	with ControlFlowSubprocess. An event will usually be produced by a prior subprocess (if it is not part of an iteration or a branching) and will .

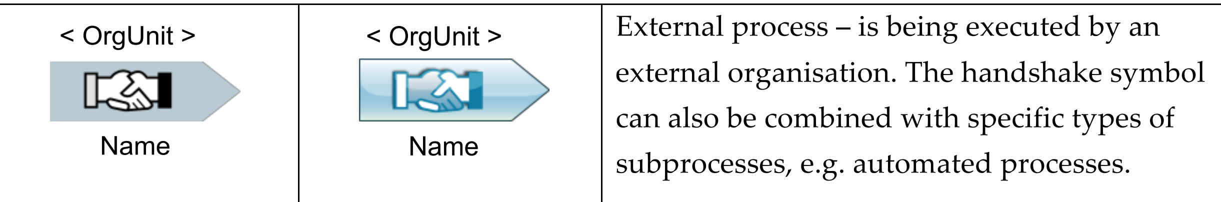
Constraints	
C4	BusinessProcess comprises all elements that constitute the corresponding process type. It is, however, sufficient to link it to the start event only.
C5	A start event type must not represent a Branch.
C6	A terminal event type must not trigger anything.
C7	An event type must not trigger more than one subsequent element at a time.
C8	An event type cannot represent a terminal and a start event at the same time.
C9	There must be no more than one start event type.
C19	There has to be at least one terminal event type within a business process model.
C20	An event type can be marked as loaded only, if it is placed before a fork.

Table 12: Comments on Event

5.3 Concrete Syntax

While regarded as irrelevant "syntactic sugar" by some, the graphical notation featured by a DSML will often be of pivotal relevance for its acceptance and usability. The graphical symbols that form the concrete syntax of the OrgML were created by a graphic artist. They are aimed at both, promoting readability and appealing, aesthetic diagrams. Currently, the graphical notation exists in two variants, "matt" and "glossy". Table 13 shows a complete dictionary of all notation elements in both variants.

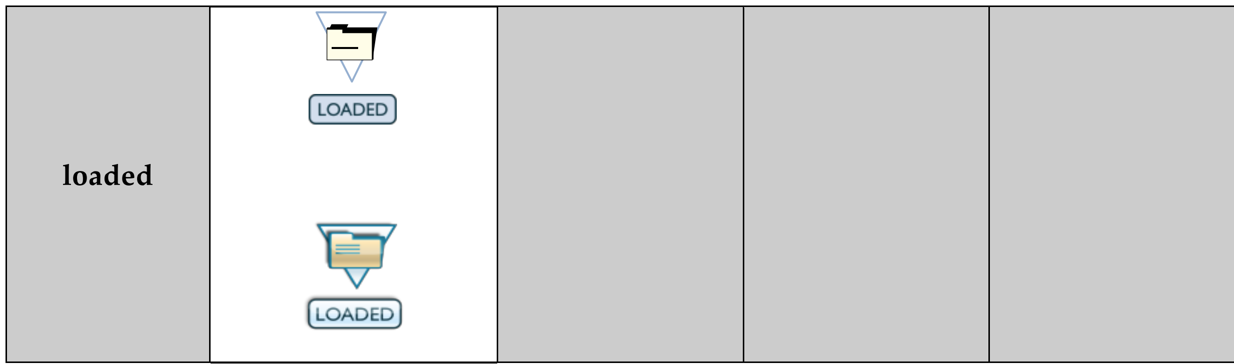
Process Symbols		
matt	glossy	Description
< OrgUnit > Name	< OrgUnit > Name	Business process type – represents instances of BusinessProcess
< OrgUnit > ID Name min: 3.30 max: 6.20 ave: 4.10	< OrgUnit > ID Name min: 3.30 max: 6.20 ave: 4.10	serves to represent subprocesses, the type of which is not specified. A subprocess ID can be represented as an optional supplement, as well as a box including minimum, maximum and average execution times. This additional information can be assigned to any subprocess type.
< OrgUnit > Name	< OrgUnit > Name	Manual subprocess – represents instances of ControlFlowSubprocess the type of which is set to #manual.
< OrgUnit > Name	< OrgUnit > Name	Automated process – represents instances of ControlFlowSubprocess the type of which is set to #auto.
< OrgUnit > Name	< OrgUnit > Name	Semi-automated process – performed by a human actor with computer support
Name	Name	Aggregated process – serves as a placeholder of a part of a business process



< OrgUnit > Name	< OrgUnit > Name	External process – is being executed by an external organisation. The handshake symbol can also be combined with specific types of subprocesses, e.g. automated processes.

Table 13: Elements of the Graphical Notation: Process Symbols

Start/Stop	START START	STOP STOP		
Change	not specified	new	modified	deleted
Time	time interval	point in time		
Notification: Human	synchronous	asynchronous, traditional	asynchronous, electronic	
Notification: Software	publish	poll		



loaded	LOADED LOADED			

Table 14: Symbols to Represent Events (upper Symbol: "matt", lower Symbol: "glossy" Variant)

Events can be differentiated in various ways: with respect to the kind of change they represent, with respect to a point in time or to duration and with respect to notification (see Table 14). The symbol at the top of each box represents the matt variant, the symbol at the bottom the corresponding glossy variant. Each event type is assigned a name that is presented below the event. In addition to that, an event type can be characterised as start or terminating event. In the current version, the combination of symbols as it is proposed in Figure 3 has not been implemented, because so far we have no sufficient evidence that it is required.

Symbols that represent exceptions are placed on top of a symbol that represents a subprocess (see Table 16). Table 15 shows the various symbols for representing exceptions, the matt variant on top, the glossy variant at the bottom.

Effect	not specified	resume	cancel
Cause	time out	human action	technical failure
Detection	Automatic	very likely	by chance

Table 15: Representation of Exceptions (upper Symbol: "matt", lower Symbol: "glossy" Variant)

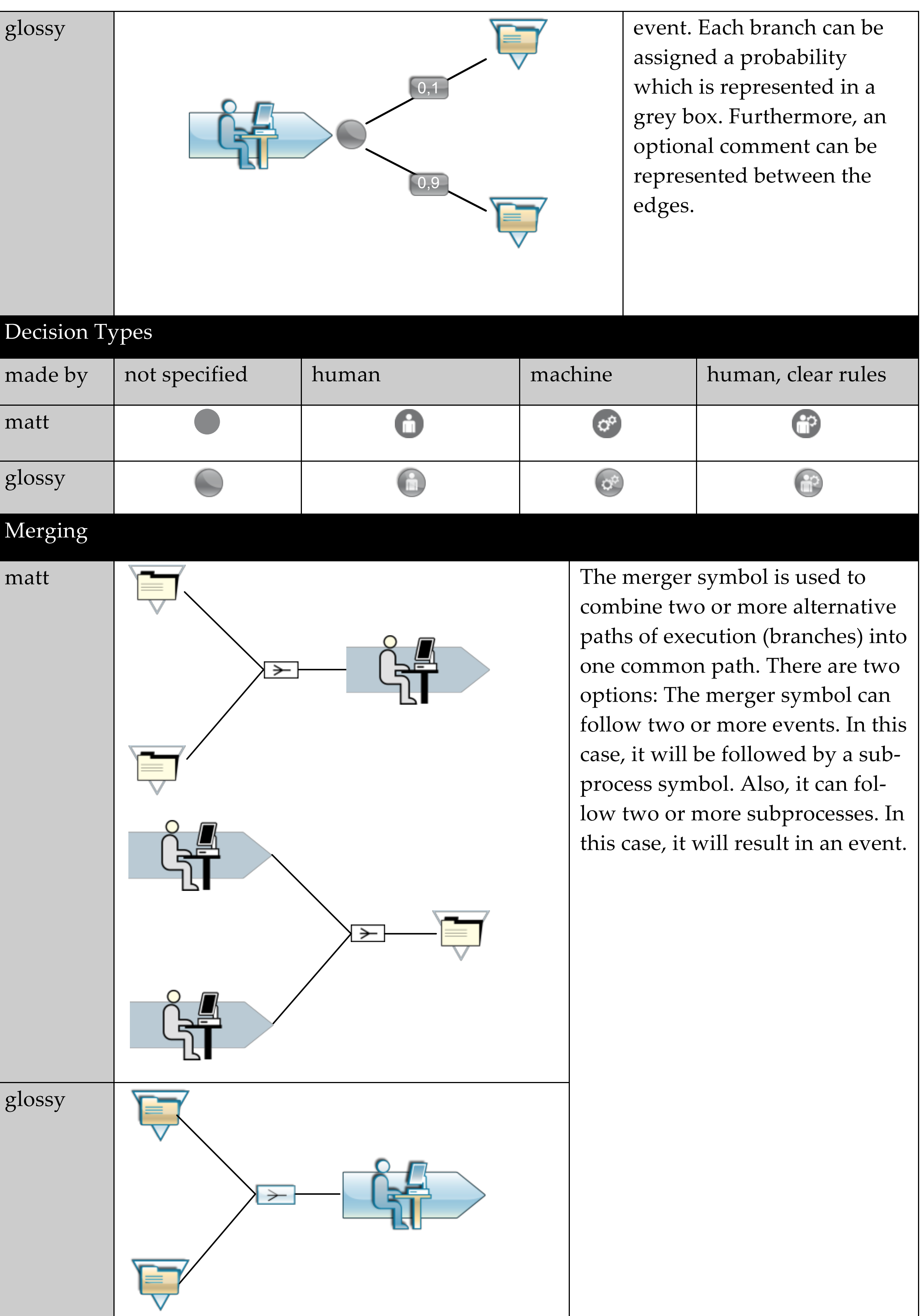
Symbols to represent effect, cause and detection can be combined, resulting in 27 different combinations. Table 16 shows a few selected combinations in the matt variant.

Description	matt	glossy
Cause: Human (customer) action **Detection**: very likely **Effect**: cancel	Customer denies <Sales Assistant > Check Credibility	Customer denies <Sales Assistant> Check Credibility
Cause: technical failure **Detection**: very likely **Effect**: resume	Connection fails <Procurement> Search Products	Connection fails <Procurement> Search Products
Cause: Time out **Detection**: automatic **Effect**: cancel	Time out <Customer> Record Order	Time out < Customer > Record Order

Table 16: Representation of Composed Exception Types

Table 17 illustrates the representation of branchings. It is the same for both variants of the notation.

Branching		
		Description
matt	0,4 0,6	A branching is attached to a subprocess symbol. It starts with a circle which serves to represent further information about the type of decision. In the given example, no decision type is specified. A branching consists of two or more branches represented as edges, each of which results in an



glossy		event. Each branch can be assigned a probability which is represented in a grey box. Furthermore, an optional comment can be represented between the edges.		
Decision Types				
made by	not specified	human	machine	human, clear rules
matt				
glossy				
Merging				
matt		The merger symbol is used to combine two or more alternative paths of execution (branches) into one common path. There are two options: The merger symbol can follow two or more events. In this case, it will be followed by a sub-process symbol. Also, it can follow two or more subprocesses. In this case, it will result in an event.		
glossy				

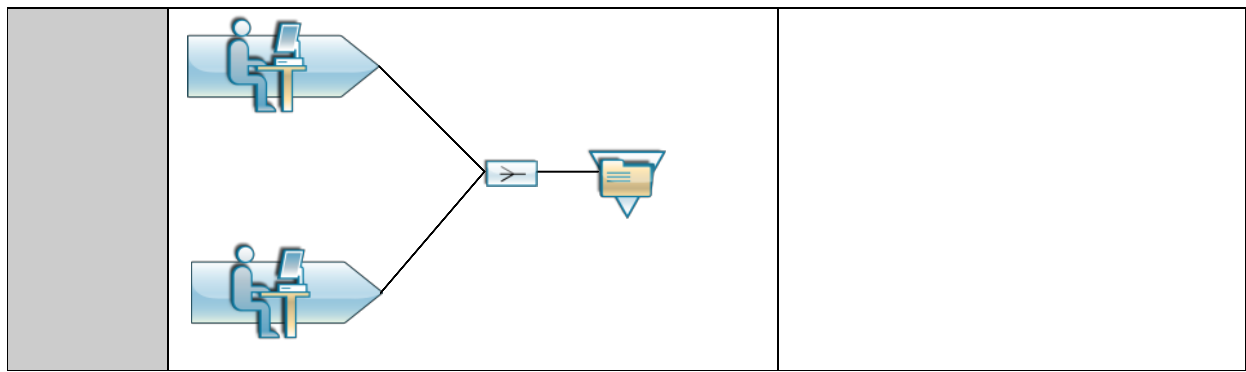
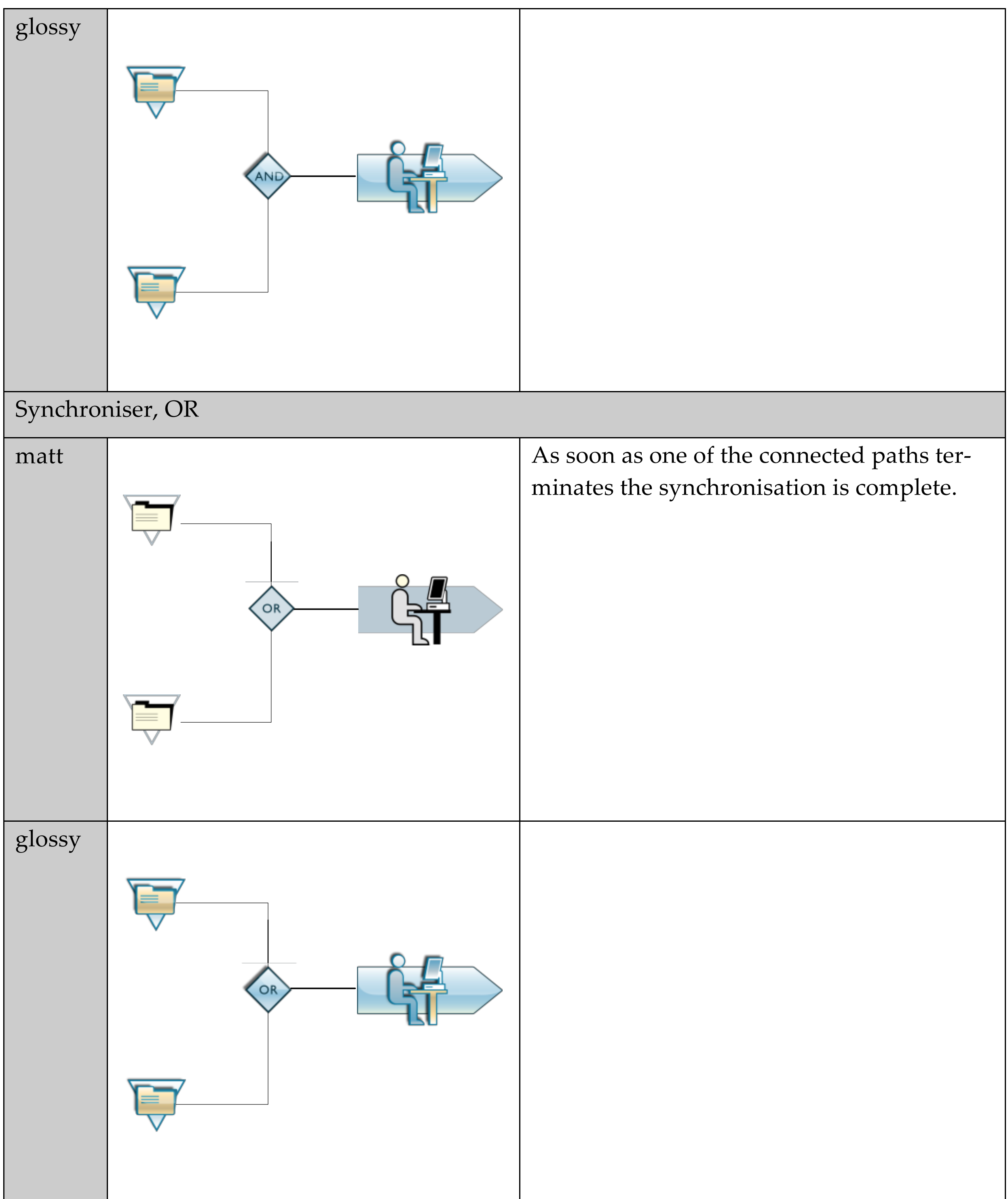


Table 17: Representation of Branchings and Mergers

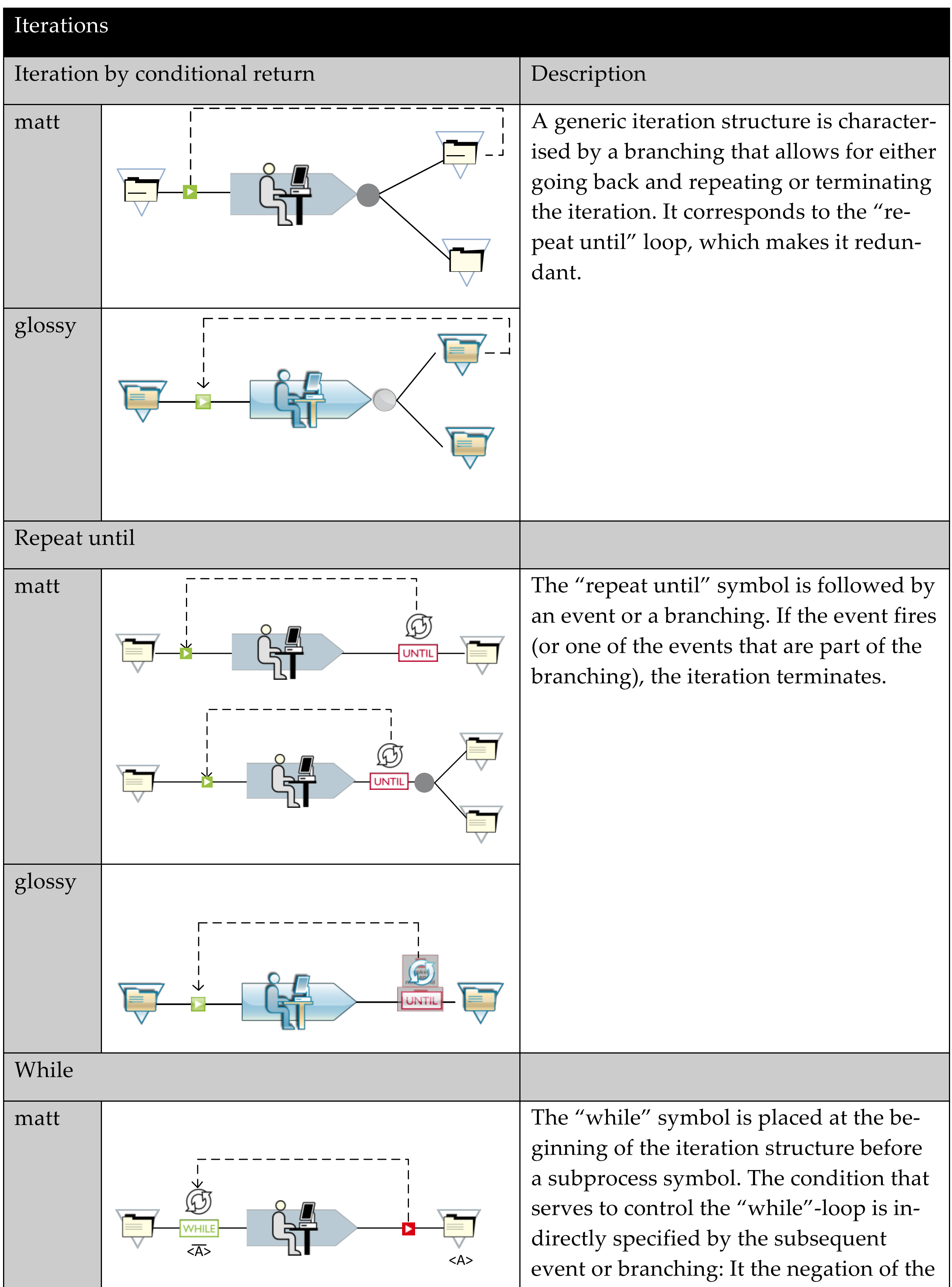
Executing paths of a business process in parallel is represented by a fork symbol that marks the beginning of a parallel execution. Finally, all parallel paths have to be synchronised using either an "And" or an "Or" synchroniser. The fork symbol is the same for both variants of the notation.

Parallel Execution		
Fork		Description
		The fork symbol follows and event and results in two or more parallel subprocesses. There is no difference between the matt and the glossy version.
Synchroniser, AND		
matt		All connected paths need to terminate to complete the synchronisation.

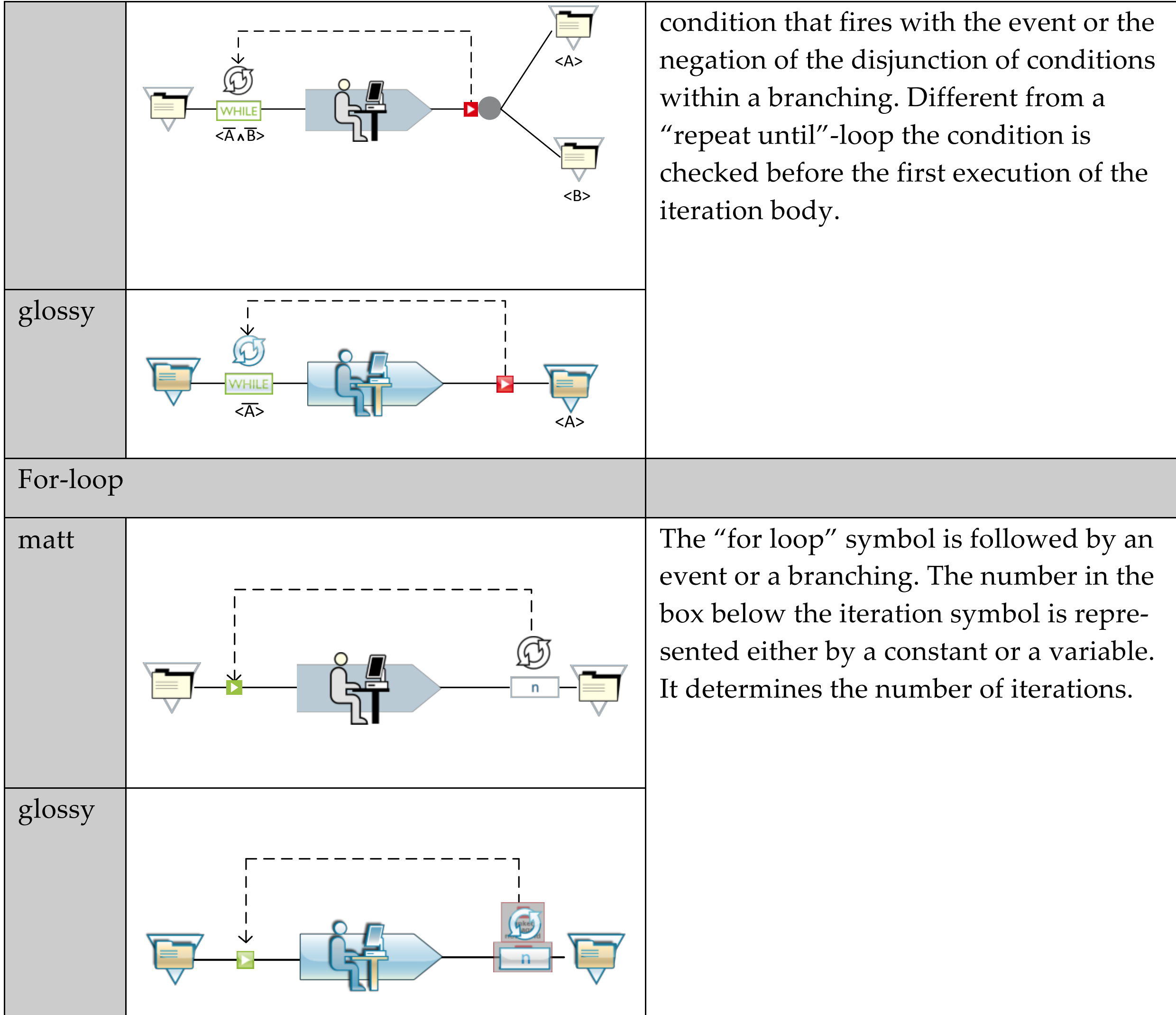


glossy		
Synchroniser, OR		
matt		As soon as one of the connected paths terminates the synchronisation is complete.
glossy		

Table 18: Representation of Synchronisers



Iterations		
Iteration by conditional return		Description
matt		A generic iteration structure is characterised by a branching that allows for either going back and repeating or terminating the iteration. It corresponds to the "repeat until" loop, which makes it redundant.
glossy		
Repeat until		
matt		The "repeat until" symbol is followed by an event or a branching. If the event fires (or one of the events that are part of the branching), the iteration terminates.
glossy		
While		
matt		The "while" symbol is placed at the beginning of the iteration structure before a subprocess symbol. The condition that serves to control the "while"-loop is indirectly specified by the subsequent event or branching: It the negation of the



		condition that fires with the event or the negation of the disjunction of conditions within a branching. Different from a "repeat until"-loop the condition is checked before the first execution of the iteration body.
glossy		
For-loop		
matt		The "for loop" symbol is followed by an event or a branching. The number in the box below the iteration symbol is represented either by a constant or a variable. It determines the number of iterations.
glossy		

Table 19: Representation of Iterations

The language includes a few advanced concepts, i.e. concepts that are not expected to be required within regular patterns of reuse.

Multiple Concurrent Instances

Sometimes the actual number of parallel paths within a parallel execution may vary. In this case, the regular representation pattern does not work anymore, since it requires representing explicitly all possible paths of execution. To overcome this limitation, the representation of multiple concurrent paths can be abstracted to one path that is assigned numbers (either constants or variables) that define the number of parallel paths. The respective numbers are placed above the fork and over the synchroniser symbols that mark the end of the parallel execution.

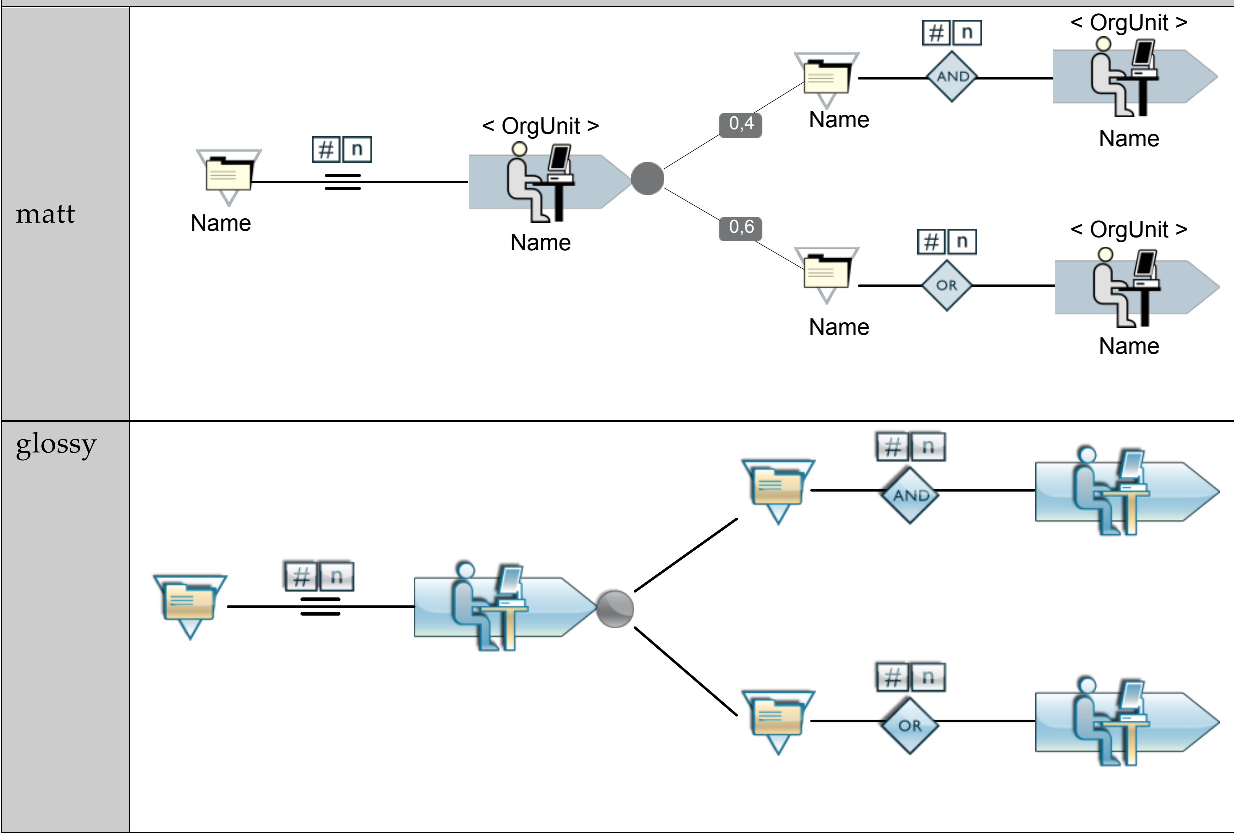


Table 20: Representation of Variable Number of Concurrent Instances

Relaxed Concurrency

As a default, two concurrent paths of execution are independent. In cases, where they may be partially dependent, the MEMO OrgML offers the concept of relaxed concurrency, indicated by a square instead of the regular fork symbol. Dependence between two parallel paths of execution is indicated by an exception symbol that is assigned – through a dotted line – to all paths that are affected. The exception corresponds to an event created in one path. Therefore the exception has to have the same name as the corresponding event.

A further aspect of relaxed concurrency relates to synchronisation: It may be that the validity of one path depends on the status of another path. If one path terminates, this may cause the termination of all other paths. Such a situation can be expressed with an "exclusive synchro-

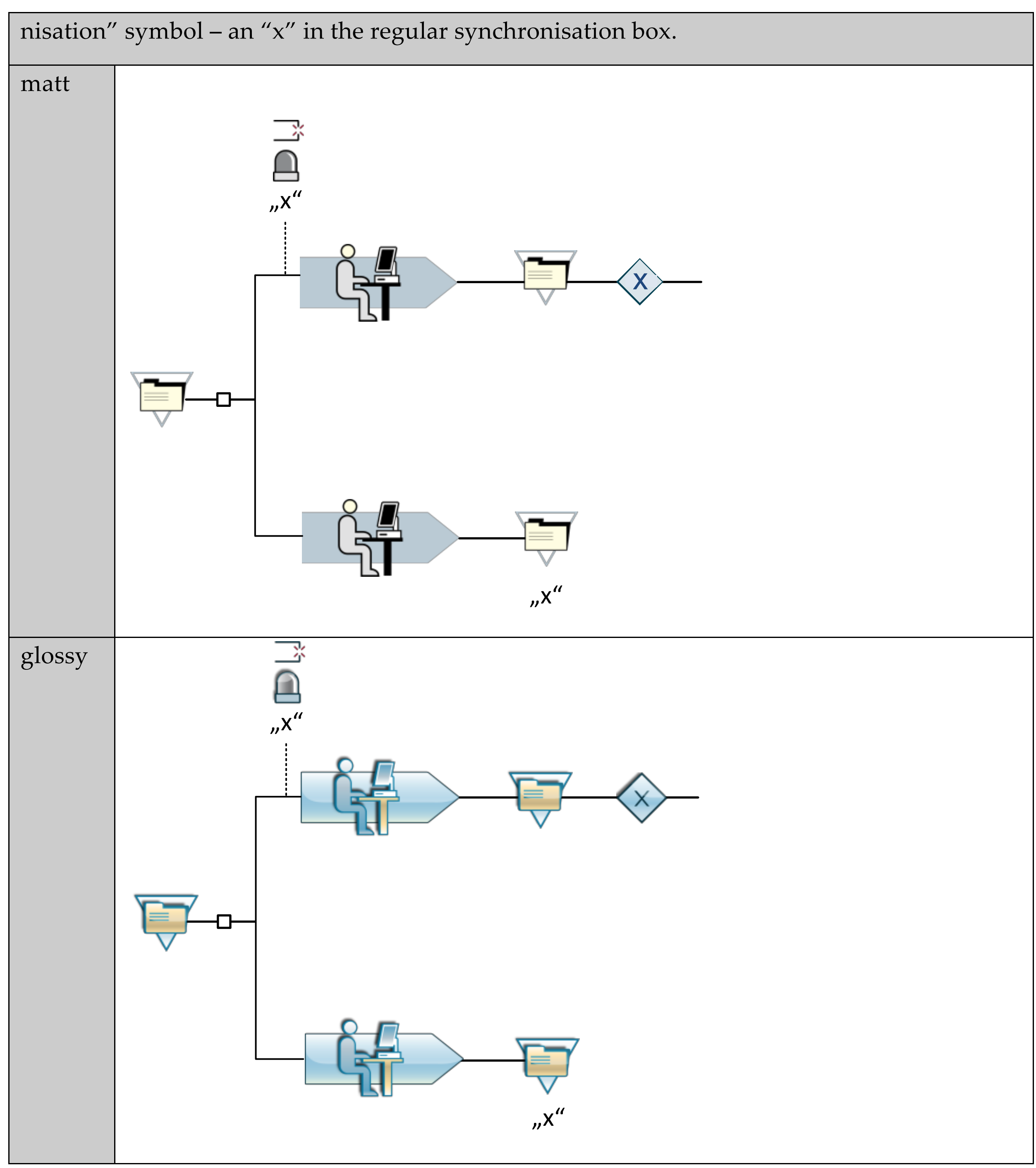
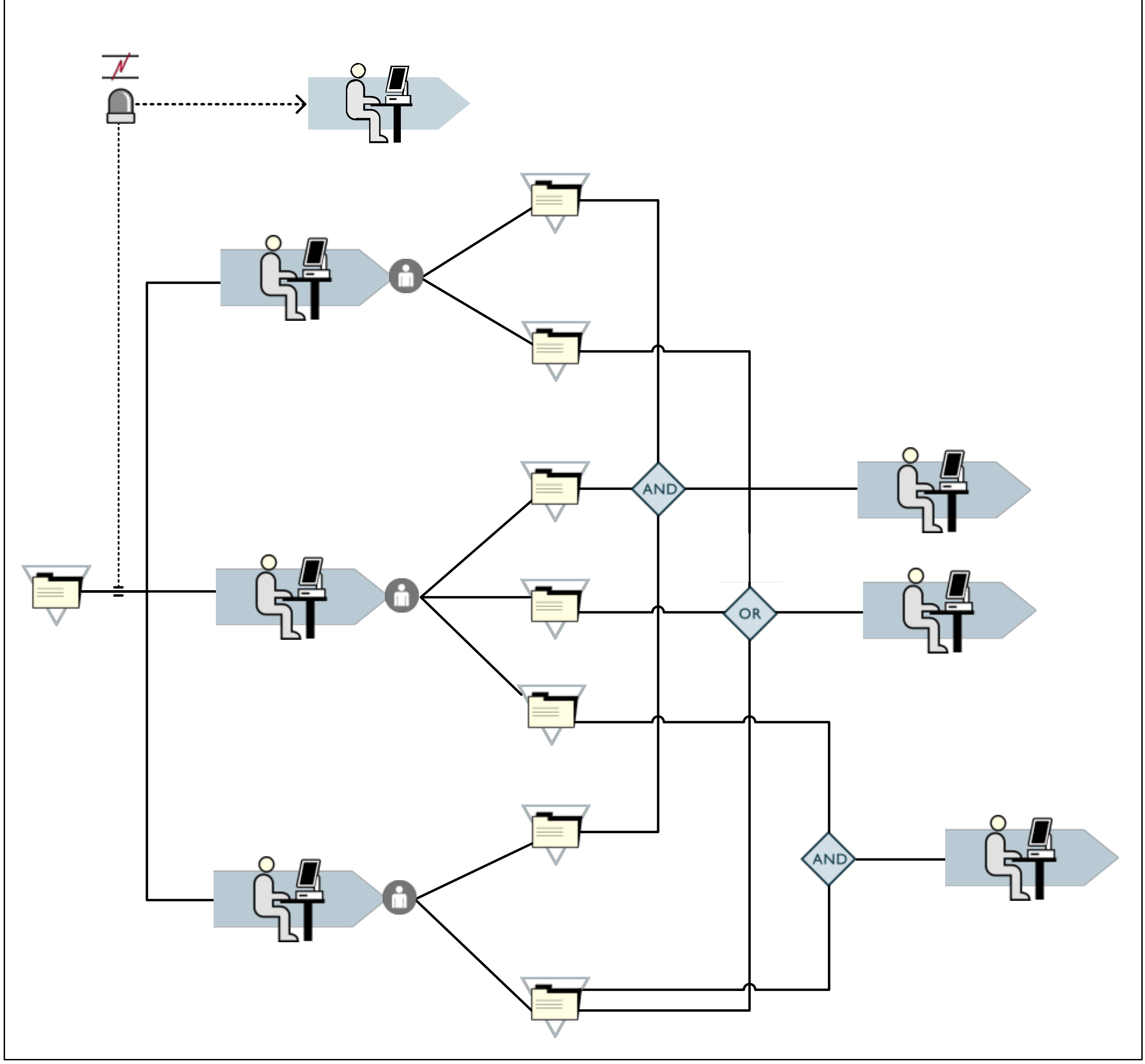


Table 21: Relaxed Concurrency, Concurrency Exception and Exclusive Synchronisation

Synchronisation Exception
A parallel execution may include many paths with multiple possible synchronisers. If only a subset of all possible synchronisers is relevant for the vast majority of process instances, the complexity of a corresponding graphical representation can be reduced by representing the relevant synchronisers only. To avoid underspecification, a synchronisation exception can be used that defines how to synchronise in those cases that are not explicitly covered. The exception symbol that represents a synchronisation exception is connected to the symbol that represents the subprocess that will be triggered after the corresponding synchronisation. The below example that corresponds to the one in Figure 40 includes constellations, the explicit synchronisers do not cover, which would result in firing the synchronisation exception which is represented at the top of the figure. The example is shown in the "matt" (top) and the "glossy" version (below).


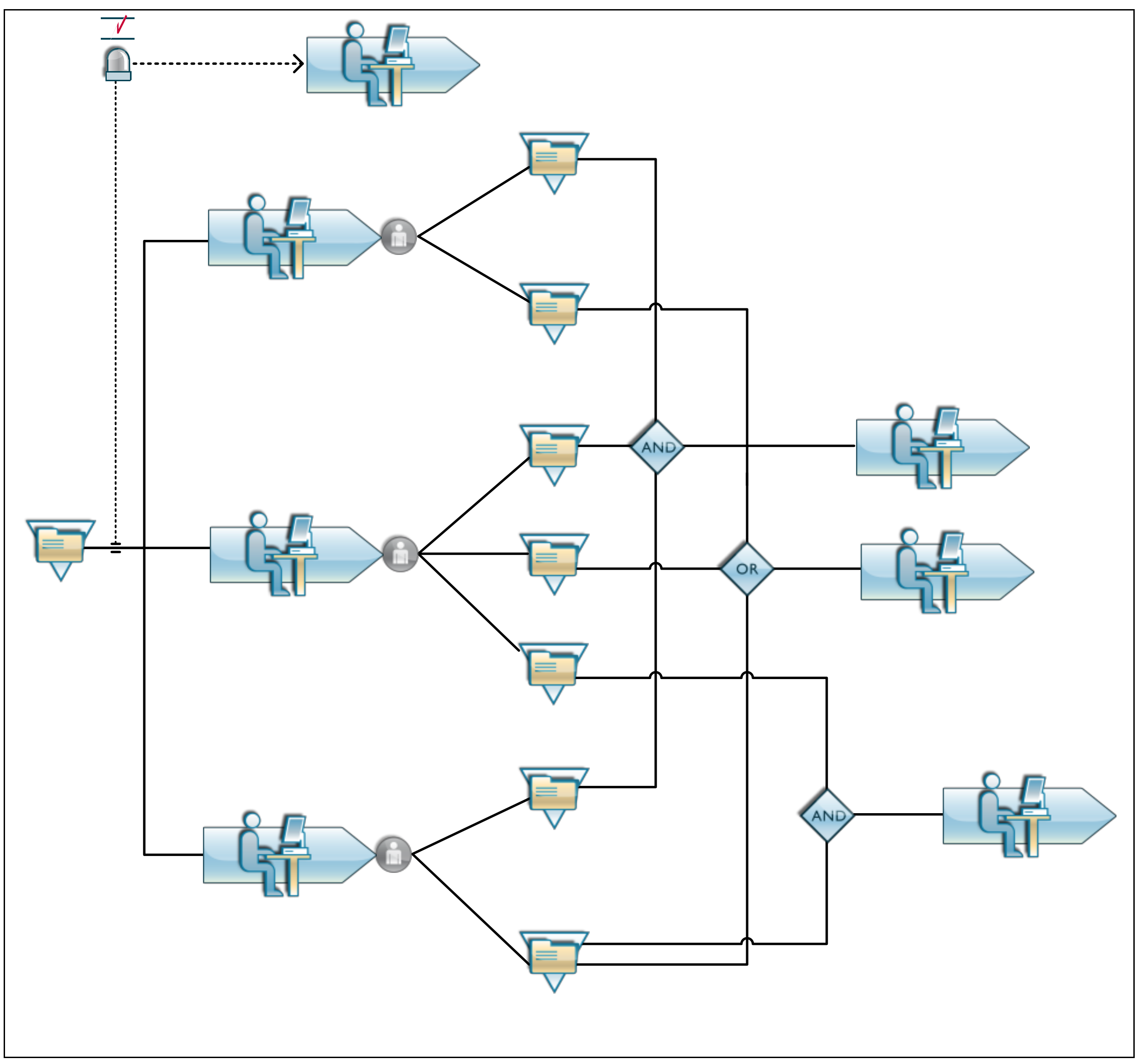


Table 22: Representation of Synchronisation Exception

Arbitrary Sequence
An arbitrary sequence is represented by an unspecified subprocess symbol and a stack of small process symbols on top. Each of the process symbols represents a subprocess. Each subprocess is assigned a name. In this representation, subprocesses cannot be distinguished with respect to their type (e.g. "automated", "manual" etc.). The symbol for "partially arbitrary sequence" includes arrows that indicate a temporal order of execution. The subprocess, the arrow starts at, has to be performed before the subprocess, the arrow points at. The sy-graphical representation of partially arbitrary sequence is illustrated on the right, the graphical representation of the arbitrary sequence on the left.

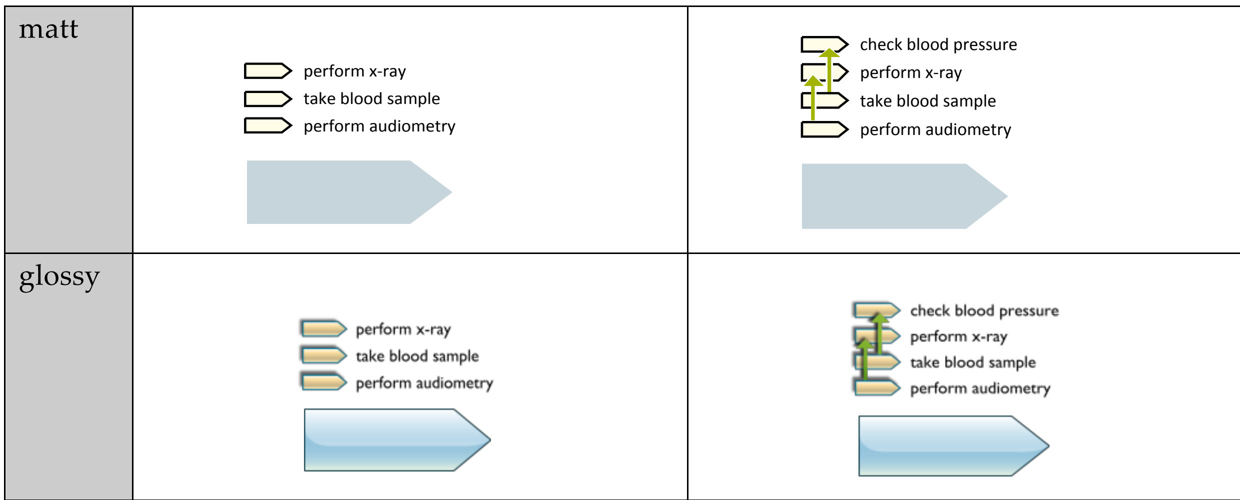


Table 23: Representation of Arbitrary Sequences

Composition/Decomposition
The main purpose of business process modelling is the specification of control flows. However, in some cases, it is sufficient to aim at a function abstraction only. A common approach in this respect is to create process composition/decomposition diagrams. For this purpose, a composition/decomposition relationship between processes is required. A composed process is represented by the same symbol that is used for representing aggregate processes. It can be decomposed into further composed processes or – finally – into a subprocess.

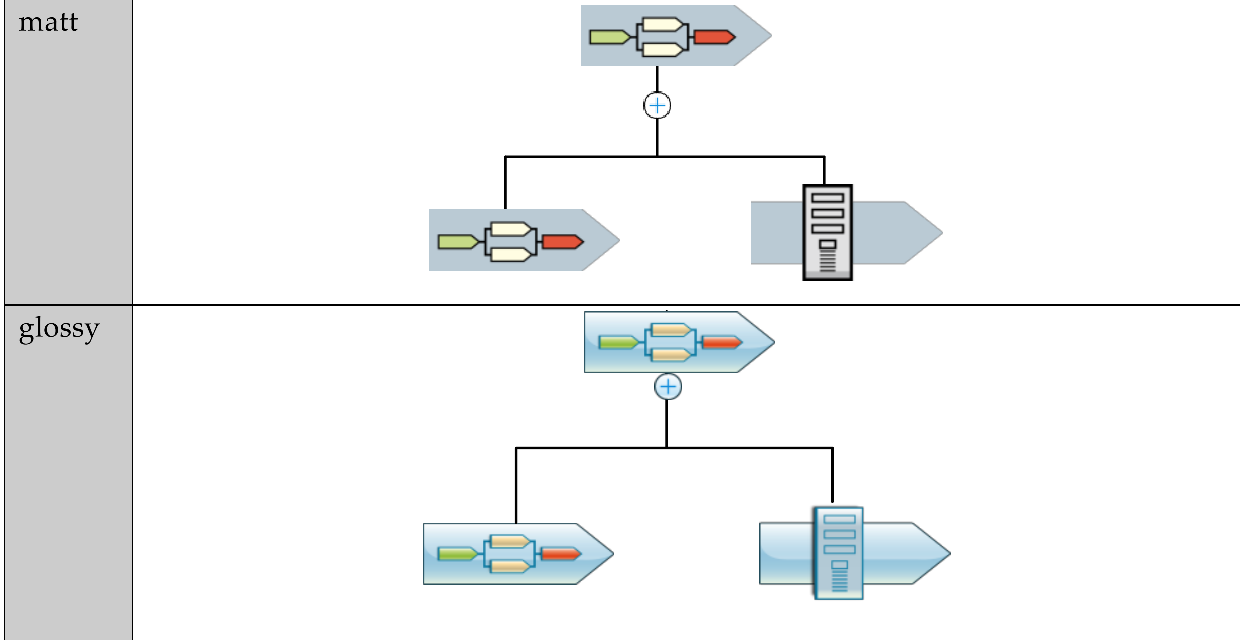
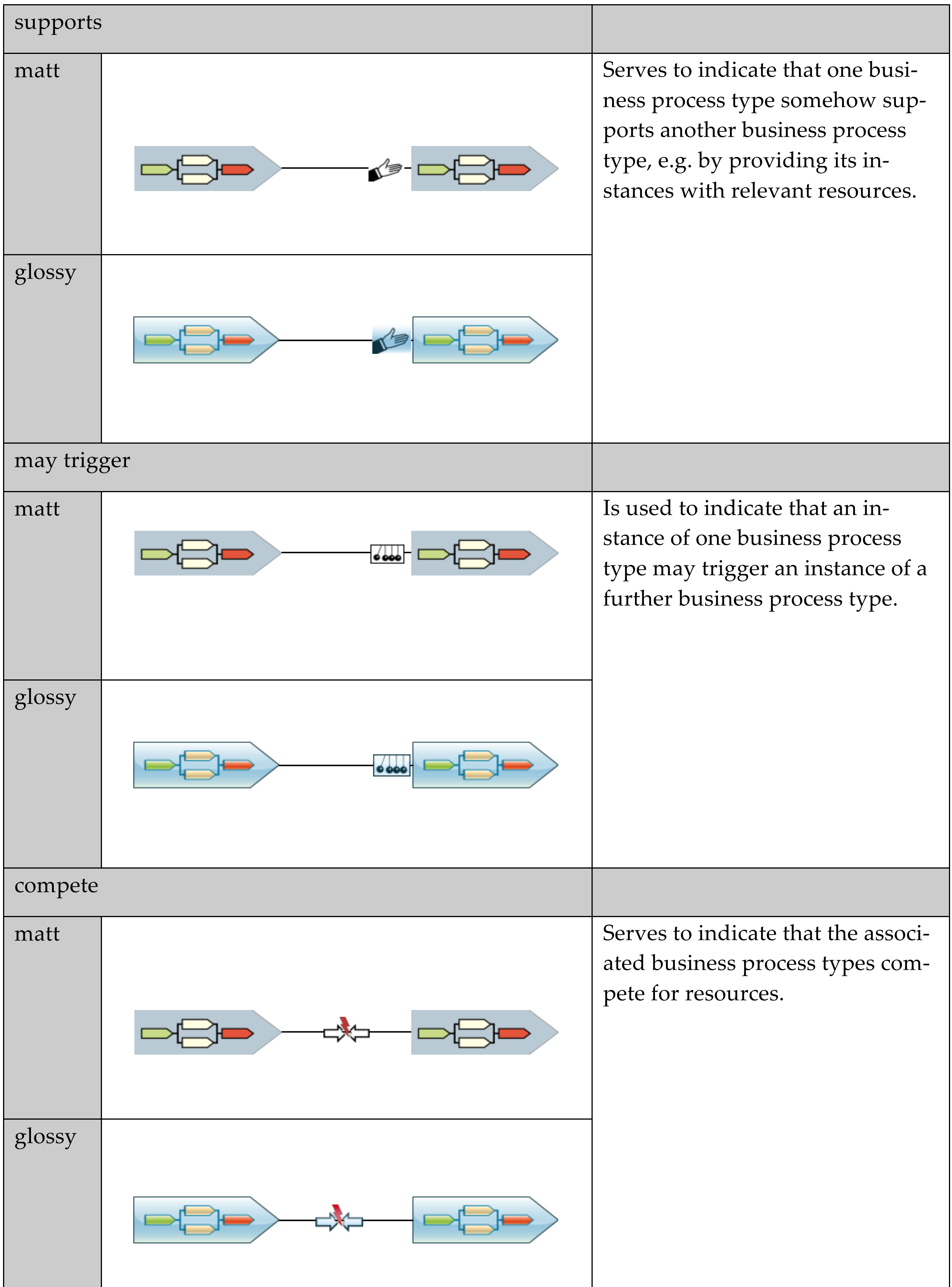


Table 24: Process Composition/Decomposition

Process "Specialisation"	
Business process types can be "specialised" into more specific types. Accordingly, subprocess types can inherit certain features to "specialised" subprocess types. Both cases are represented by the same arrow that points at the super concept.	
matt	
glossy	

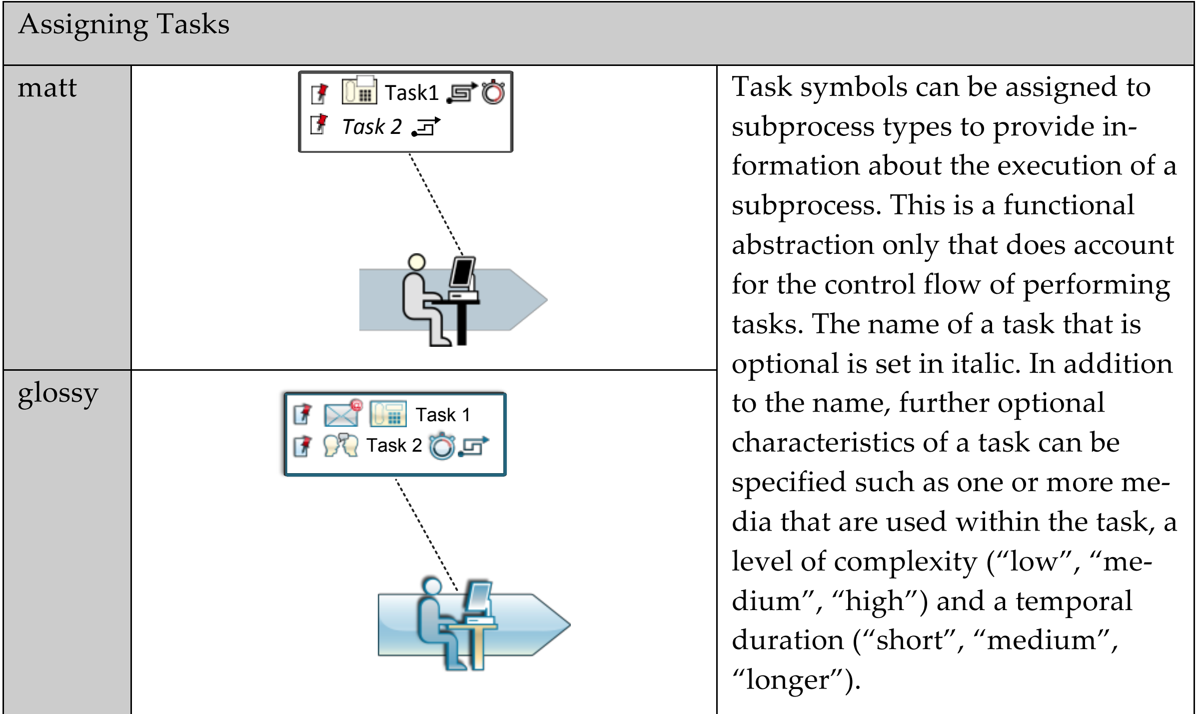
Table 25: Symbols for Representing Process Specialisation

Associations between Business Processes		
Similarity		Description
matt		Serves to indicate that two business process types are similar. Since no formal specification of similarity is provided, an additional comment should clarify the criteria the associated business process types have in common.
glossy		



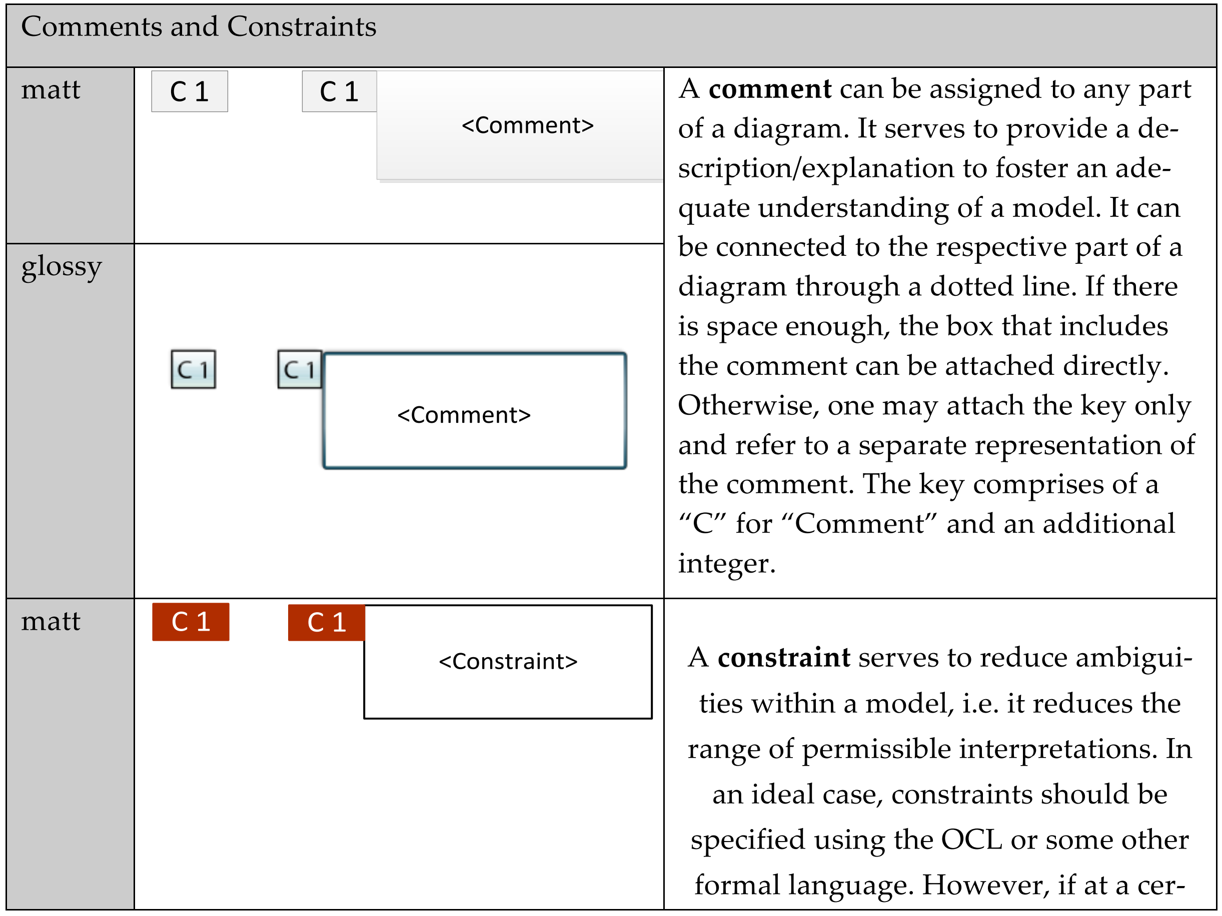
supports		
matt		Serves to indicate that one business process type somehow supports another business process type, e.g. by providing its instances with relevant resources.
glossy		
may trigger		
matt		Is used to indicate that an instance of one business process type may trigger an instance of a further business process type.
glossy		
compete		
matt		Serves to indicate that the associated business process types compete for resources.
glossy		

Figure 49: Symbols to represent associations between business process types

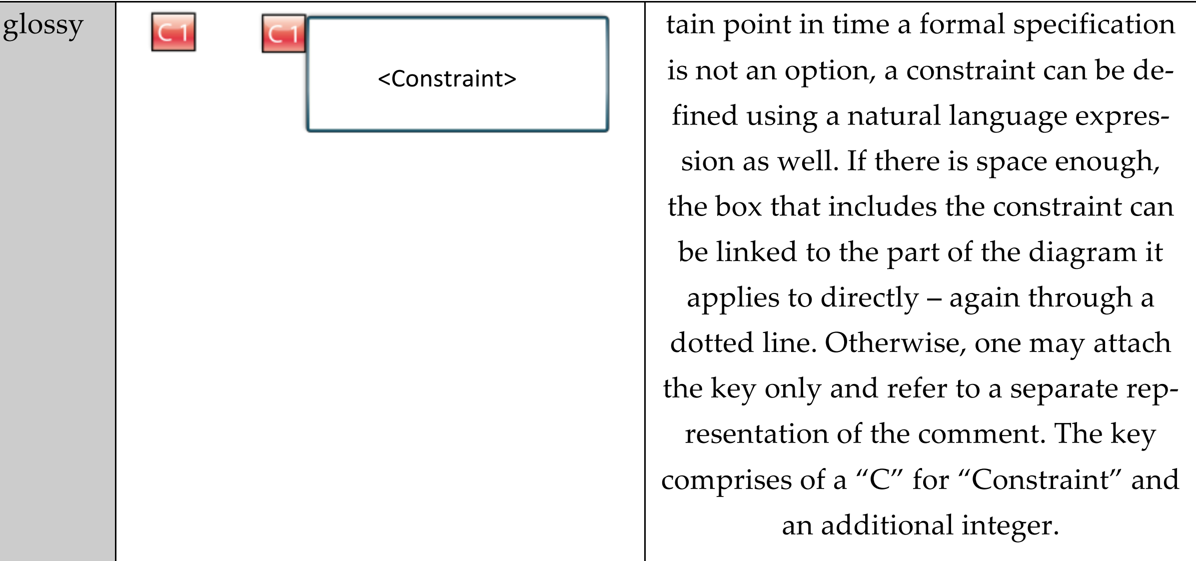


Assigning Tasks		
matt		Task symbols can be assigned to subprocess types to provide information about the execution of a subprocess. This is a functional abstraction only that does account for the control flow of performing tasks. The name of a task that is optional is set in italic. In addition to the name, further optional characteristics of a task can be specified such as one or more media that are used within the task, a level of complexity (“low”, “medium”, “high”) and a temporal duration (“short”, “medium”, “longer”).
glossy		

Table 26: Assigning Tasks to Processes

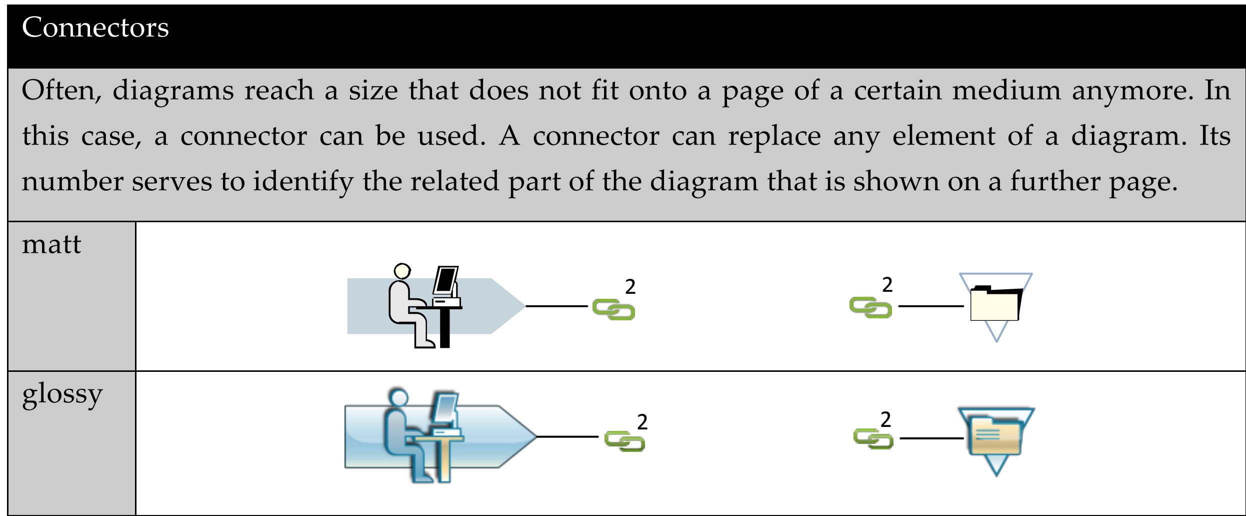


Comments and Constraints		
matt		A **comment** can be assigned to any part of a diagram. It serves to provide a description/explanation to foster an adequate understanding of a model. It can be connected to the respective part of a diagram through a dotted line. If there is space enough, the box that includes the comment can be attached directly. Otherwise, one may attach the key only and refer to a separate representation of the comment. The key comprises of a “C” for “Comment” and an additional integer.
glossy		
matt		A **constraint** serves to reduce ambiguities within a model, i.e. it reduces the range of permissible interpretations. In an ideal case, constraints should be specified using the OCL or some other formal language. However, if at a cer-



glossy		tain point in time a formal specification is not an option, a constraint can be defined using a natural language expression as well. If there is space enough, the box that includes the constraint can be linked to the part of the diagram it applies to directly – again through a dotted line. Otherwise, one may attach the key only and refer to a separate representation of the comment. The key comprises of a "C" for "Constraint" and an additional integer.

Figure 50: Comments and Constraints



Connectors	
Often, diagrams reach a size that does not fit onto a page of a certain medium anymore. In this case, a connector can be used. A connector can replace any element of a diagram. Its number serves to identify the related part of the diagram that is shown on a further page.	
matt	
glossy	

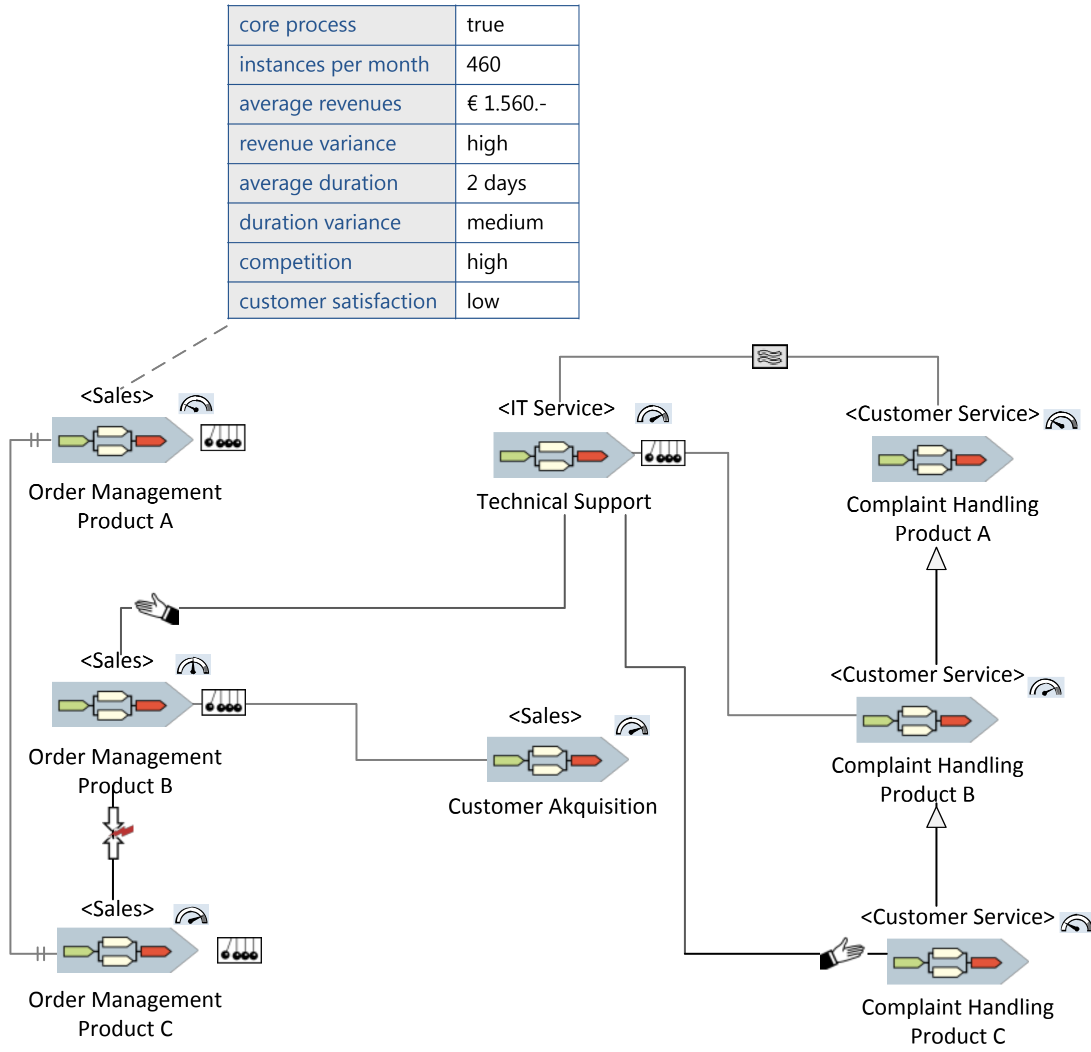
Figure 51: Connector

6 Examples

The following examples illustrate the use of the MEMO OrgML for modelling business processes through various small case studies. The diagrams were created using both the “matt” and the “glossy” notation. Some examples are supplemented by a modified variant that omits the use of events.

6.1 Business Process Map

On a high level of abstraction, a business process map can be used to give a “ballpark view” of a company’s business process types. On the one hand, it allows for illustrating relationships between business process types. On the other hand, it can be used to represent features that are characteristic for a selected process type or for comparing business process types with respect to certain features. Figure 52 shows an example business process map with additional information on one selected process type.



core process	true
instances per month	460
average revenues	€ 1.560.-
revenue variance	high
average duration	2 days
duration variance	medium
competition	high
customer satisfaction	low

Figure 52: Business Process Map

6.2 Process (De-) Composition Diagram

A process composition/decomposition diagram serves to illustrate function composition of processes. It may be useful in cases where control flow aspects can be faded out. To support analysis, the process symbols could be supplemented with performance symbols or with references to organisational units.

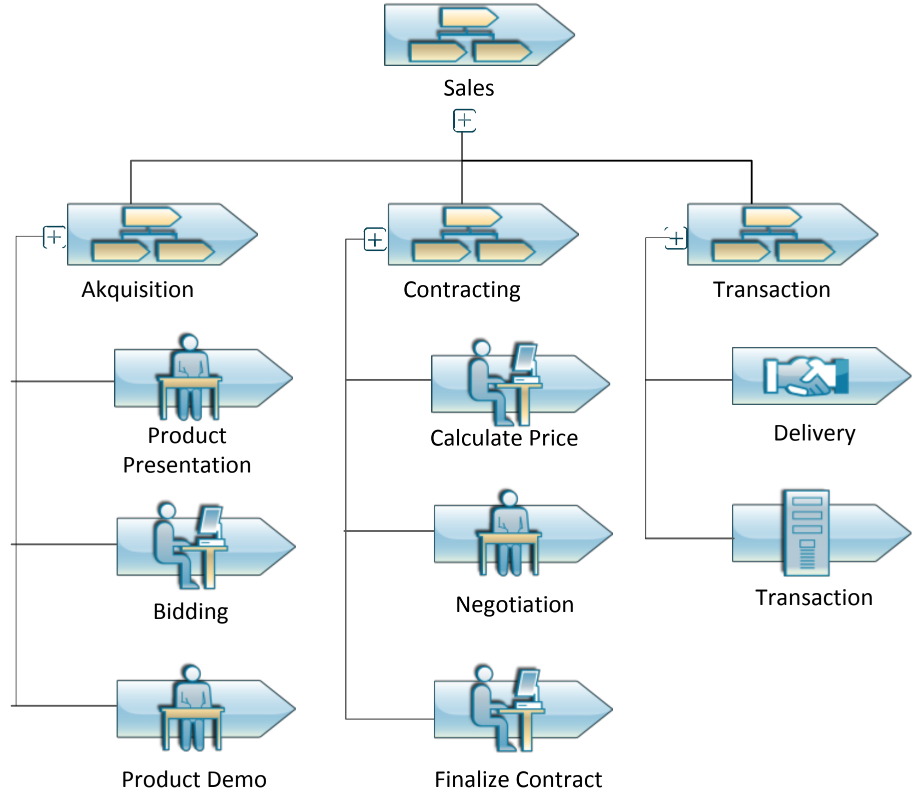


Figure 53: Illustration of Process (De-) Composition Diagram

6.3 Process Inheritance Diagram

A process inheritance diagram allows for representing specialisation relationships between subprocess types. They are based on a relaxed conception of specialisation that is restricted to inheriting tasks (which may be related to positions) and methods (which have to be associated with a class). Therefore, manual subprocesses must not inherit from automated subprocesses – et vice versa. Process inheritance diagrams can be helpful with respect to maintaining larger collections of business process types that share similar subprocesses. Designing inheritance diagrams should be done with great care: Only if an inheritance relationship is regarded as invariant, it should be introduced as such. With respect to safe and convenient maintenance of business process libraries, process inheritance diagrams represent only an intermediate step. After a task in a super subprocess type was added or changed, the affected business process types have to be detected and accordingly updated. The semantics for automating these operations has not been defined yet.

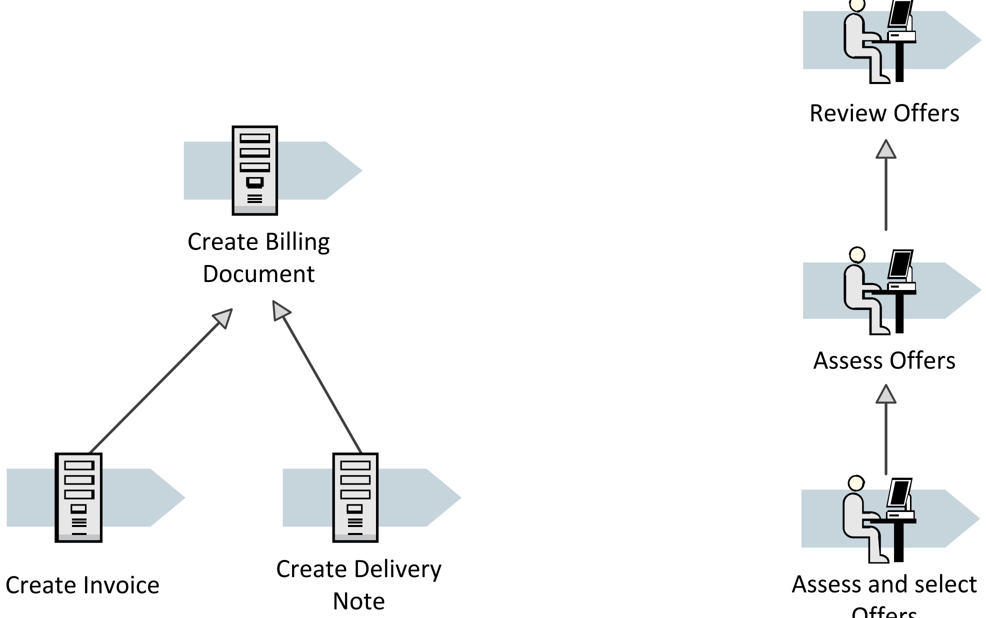


Figure 54: Illustration of Process Inheritance Diagram

6.4 Order Management

Order management serves as a classical example for business processes. The first case represents an order management process with a company that sells printers. The product line ranges from low budget printers that are sold at less than € 100 a piece to high performance printers with a price tag of € 5.000 and more. During the last months the number of customer complaints has increased. Therefore, the current process is to be analysed and – if necessary – revised.

6.4.1 Current Process

Figure 55 shows the current situation at a glance. If execution time is a core objective, this solution may be unsatisfactory, because it requires sequential testing of availability and delivery capability. The diagram also illustrates the use of comments and the representation of tasks.

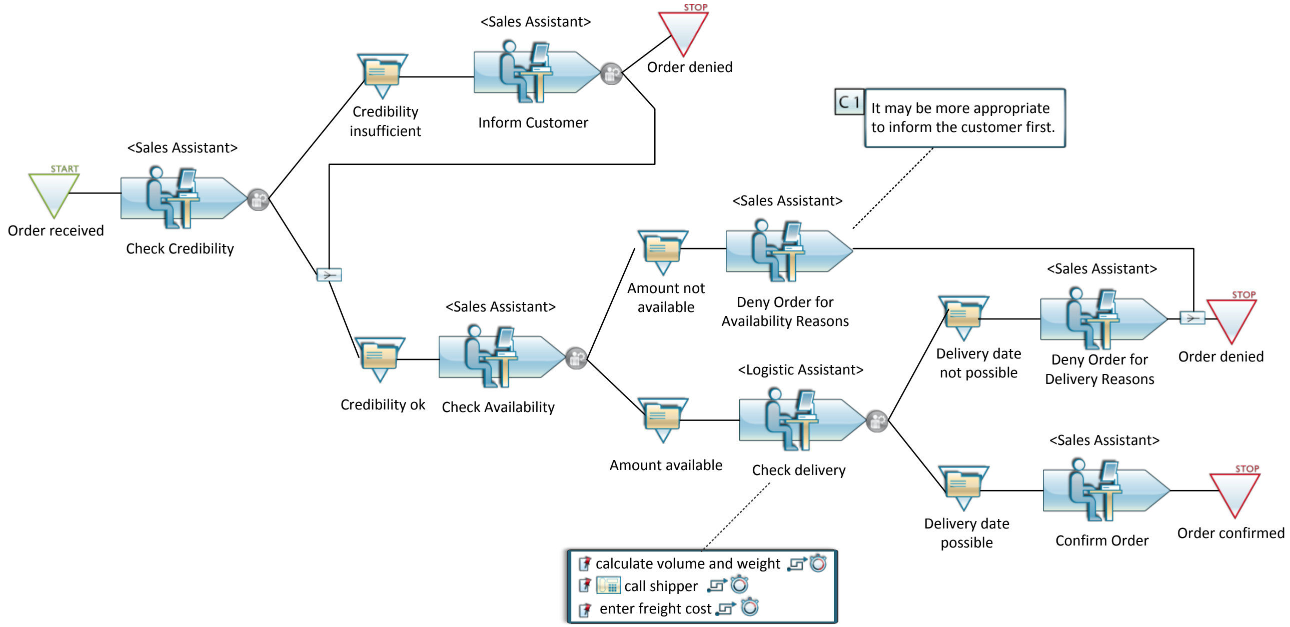


Figure 55: Existing Order Management Process

6.4.2 Alternative Version

The alternative version puts emphasis on reducing execution time by defining the parallel execution of both tests. It comes with the downside of increasing costs. The frequency of denials and the priority of process goals (cost reduction vs. reduction of execution time) have to be accounted for to make an appropriate choice.

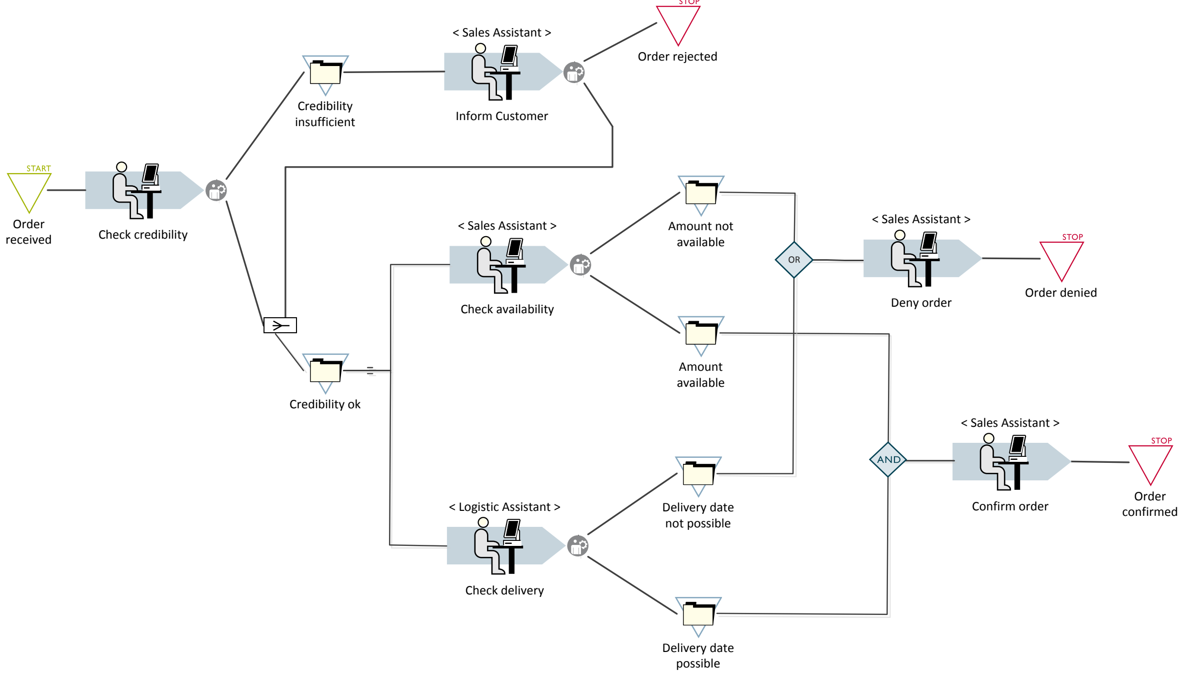


Figure 56: Alternative Design of Order Management Process

6.5 Procurement (ECOMOD)

The following examples result from the research project ECOMOD that was aimed at designing and documenting reference business processes for electronic commerce. Originally, they were designed in a previous graphical notation. The example in Figure 56 shows a high-level representation of a procurement process that includes various aggregated processes.

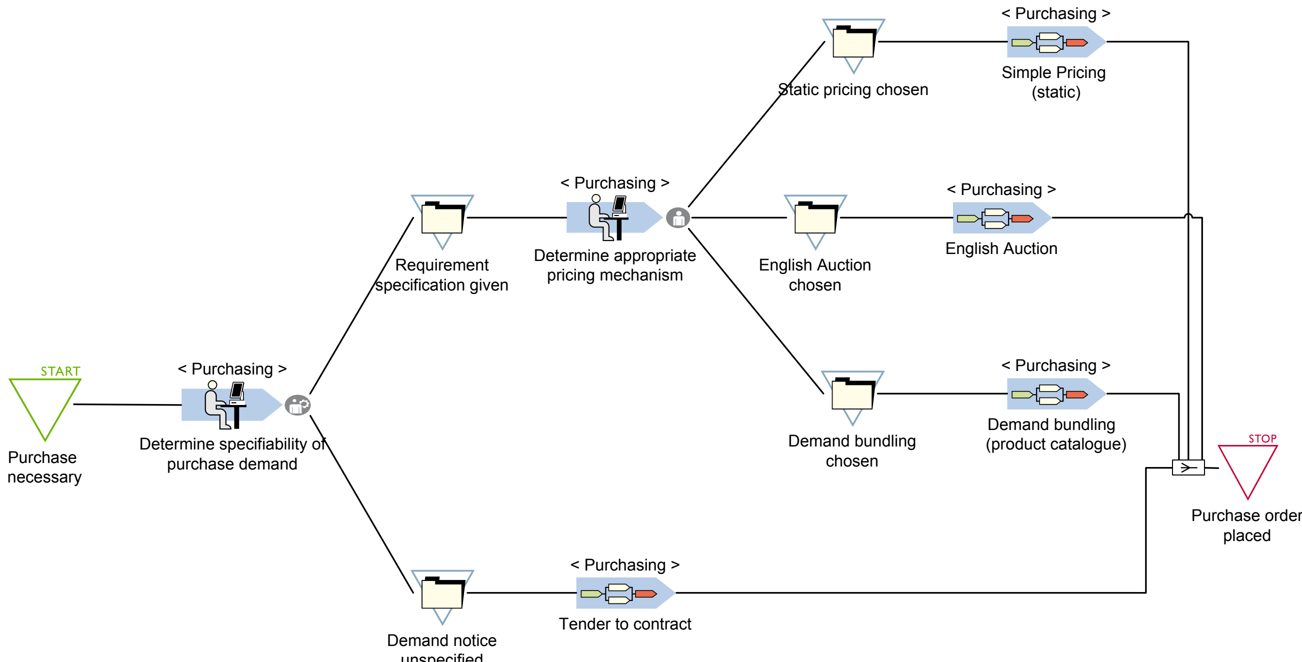


Figure 57: High-Level Process

This process shown in Figure 57 aims at evaluating a number of given offers. The evaluation process starts with determining the number of responsible persons. This list of persons is assumed to be derived automatically, e.g. on basis of the price or the type of the product. A workflow supporting system subsequently determines the first person to evaluate the offer. After his evaluation the next person is determined automatically. This process is repeated as long as there are other responsible persons on the list. When all evaluations have been performed the final evaluation is determined, which can result in one offer selected or in no appropriate offer found.

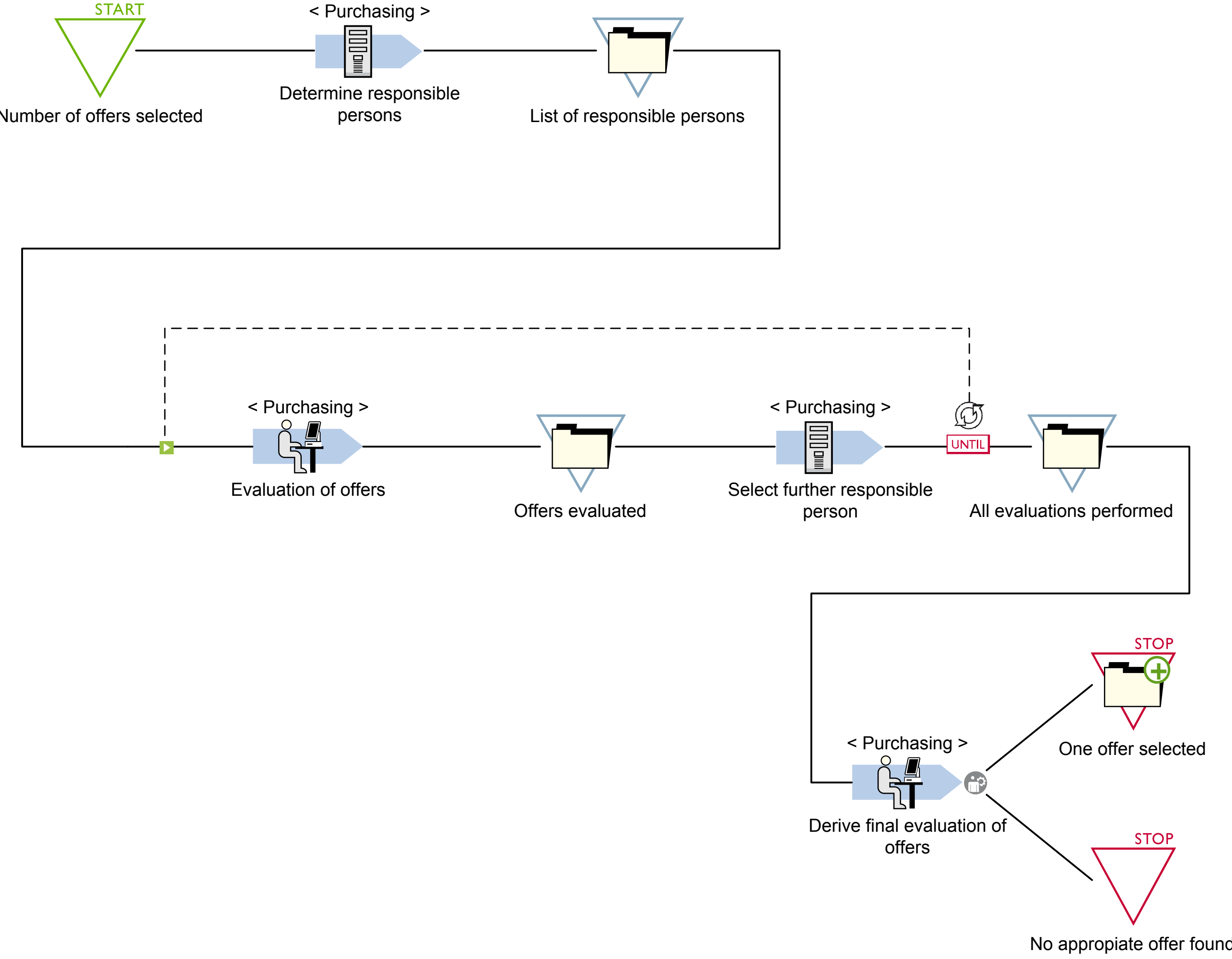


Figure 58: Multi-Person Evaluation of Offers

Figure 59 shows a traditional business process where a customer order is processed widely manually. It is a variant of the example in Figure 55.

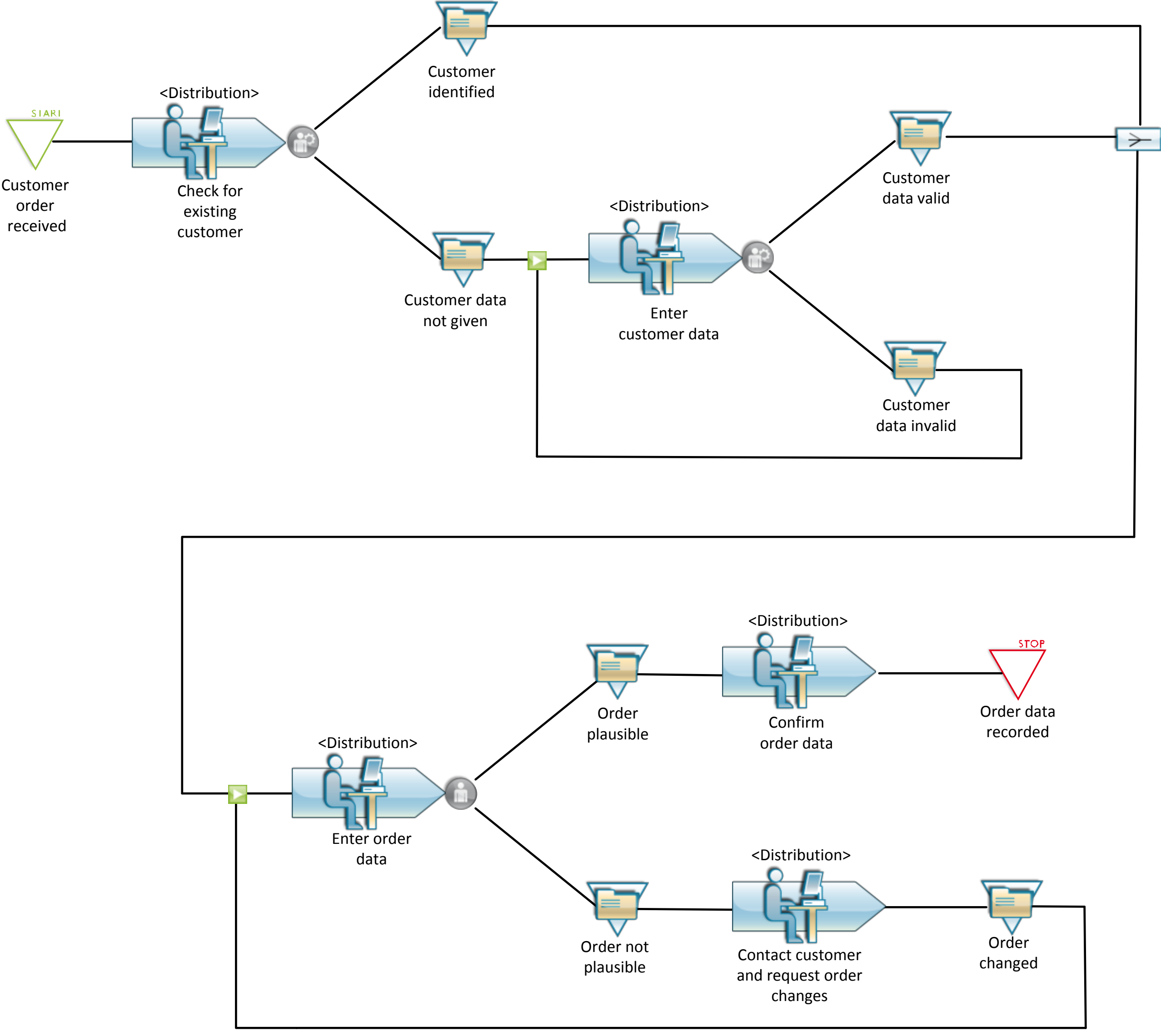


Figure 59: Manual Order Entry

6.6 "Light" Notation

Events have an important function for defining the control flow. However, representing them explicitly increases the size of a corresponding diagram and may thus contribute to reducing its comprehensibility. Therefore, business process diagrams could also be represented without events – except for the start event and the terminal event(s). Such a "light" version of the notation has not been implemented yet. It would not require changing the meta model. Instead, a model editor would have to implement a "light" drawing mode, where intermediate events would be included implicitly and would not be shown. If required, names of event types could be assigned to the edges that replaced event types. This would make particularly sense in the case of branchings. Figure 60 and Figure 61 illustrate how the diagrams shown in Figure 55 and Figure 56 would look like in the "light" notation.

Examples

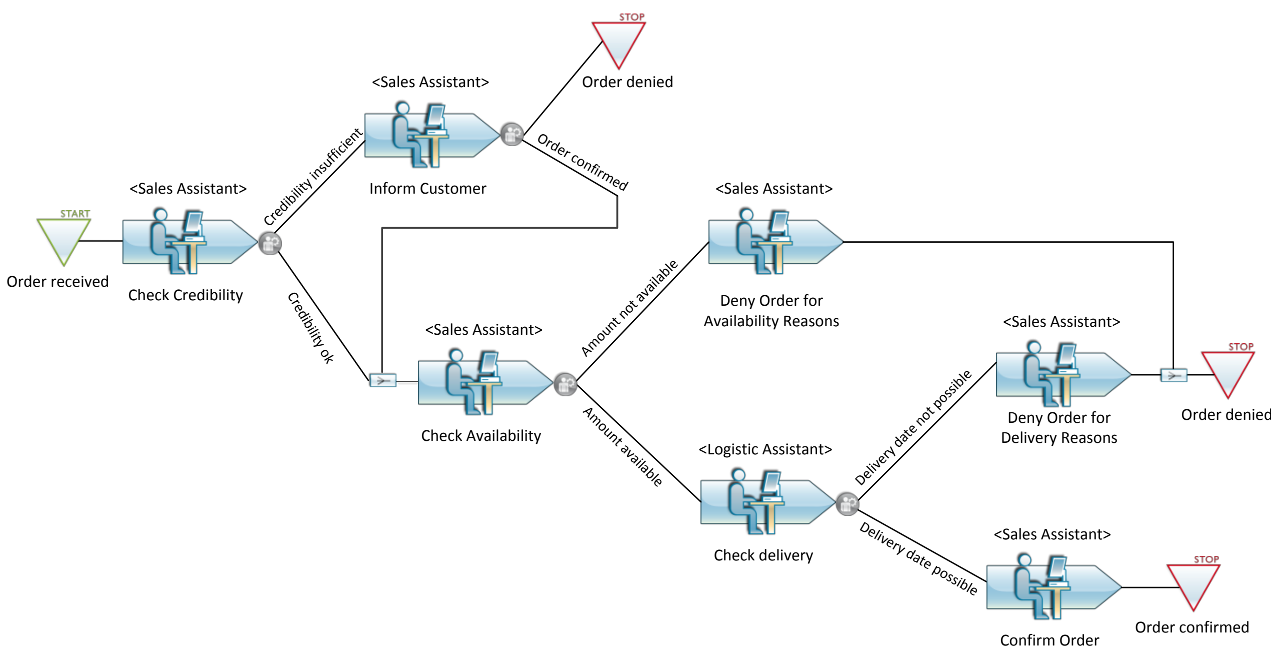


Figure 60: "Light" Notation - Example Diagram 1

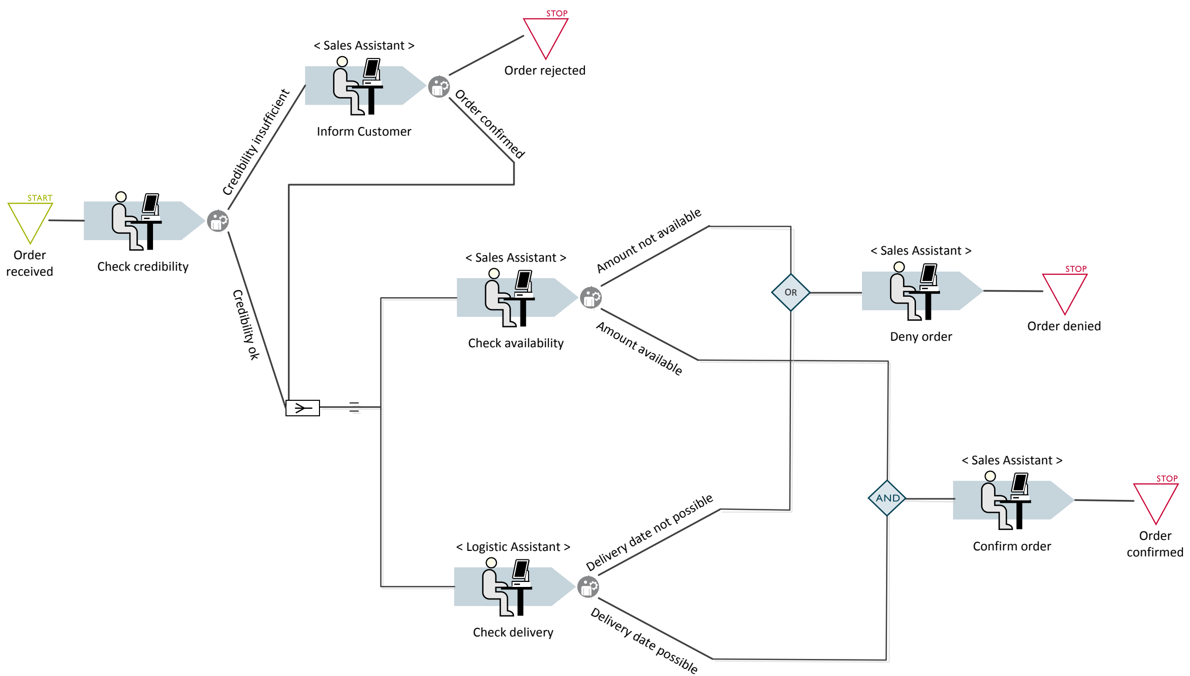


Figure 61: "Light" Notation - Example 2

It will be one topic of our future research to analyse under which conditions it is preferable to do without the explicit representation of intermediate event types.

7 Evaluation of the MEMO OrgML

The evaluation of a linguistic artefacts is a specific challenge (for a discussion of particular problems see Frank 2006a). It may seem plausible to aim at an empirical evaluation: A DSML is supposed to be an instrument that should serve its prospective users as a tool. An empirical test could be used to find out, to which extent this is the case. However, an empirical investigation of languages faces serious obstacles. Empirical studies that go beyond looking at small cases are usually not feasible, since they would require introducing modelling tools and training a substantial number of representative users. Even more important are epistemological problems: The benefit to be expected from a language depends on previous experience, attitude – i.e. the linguistic predetermination. In addition to that, it may take time for prospective users to learn and appreciate (or dislike) a language. Most of the corresponding factors are highly contingent, which makes it almost impossible to develop a clear evaluation based on an empirical study that is restricted to a point in time (for a more comprehensive discussion see Frank 2006b, p. 22 ff.). Nevertheless, it is certainly useful to test a DSML with possible users – to get a better idea of how they perceive it, whether the level of detail seems appropriate etc. We performed tests like that during the last years with our students and they have contributed to the development of the MEMO languages. However, they are hardly sufficient for evaluating the language. The evaluation we performed is based on the requirements, the OrgML should satisfy (Frank 2011b). It includes both, the part of the OrgML intended for modelling organisational structures (Frank 2011c) and the part on process modelling presented in this report. For some of the requirements, it can be decided with substantial confidence whether they are satisfied or not. The fulfilment of other requirements depends on contingent factors, i.e. it cannot be determined without accounting for further detail. The degree to which a requirement is satisfied is expressed with the symbols "+" (satisfied), "o" (satisfied to some degree) and "-" (not satisfied). The symbol "c" (contingent) indicates that the fulfilment of a requirement depends on contingent factors.

ID	Requirement		Comment
F1	The specification of a modelling language should include a precise and complete specification of its syntax. In an ideal case, this will be a formal specification. In any case, the syntax specification should allow a human to clearly decide whether a specific model is syntactically correct or not.	**+**	The abstract syntax and semantics of the DSML are specified in the meta model and additional formal constraints.
F2	In order to support the implementation of corresponding modelling tools, the specification language should correspond to languages used for software design.	**+**	The MEMO MML accounts for the semantics of prevalent programming languages.
F3	The rules defining the semantics of a modelling language should be suited to clearly guide prospective	**o**	While elaborate definitions of the core language concepts of the lan-

	users with the construction of appropriate models and their adequate interpretation. These rules should be formalised, if this does not compromise the intended meaning.	c	guage are provided, there is still a certain degree of freedom of how to apply them to develop models of organisation structure.
F4	The modelling language should feature concepts that foster a high level of abstraction to support model integrity and reuse.	-	There are only a few high-level concepts that can be used to foster model integrity, e.g. defining the max. number of levels. In addition to that, common auxiliary types promote integrity and reuse. The language lacks further abstraction concepts such as generalisation/specialisation.
U1	The concepts of a modelling language should correspond to concepts prospective users are familiar with. That recommends reconstructing existing terminology. Furthermore, it recommends using graphical symbols that are suited to illustrate the corresponding concepts' meaning.	+ c	The language design was guided by existing terminology. However, since the terminology is not used in a uniform way, there may be users who do not feel familiar with the concepts at first. The expressiveness of the graphical notation has been subject of various revisions. Nevertheless it cannot be expected to be perceived homogeneously be different groups of users.
U2	To overcome the conflict between convenience of use and simplicity, a language should provide a core of basic concepts that are sufficient for creating simple models.	+	Using core concepts such as OrganisationalUnit, Position and "is part of" relationships will be sufficient to create simple models
U3	The modelling language should allow for building models on various levels of detail and abstraction. A modeller should not be forced to specify detail he does not need.	o	Organisation models can be built on different levels of abstraction and detail, e.g. by using or omitting local types or by using categories. Furthermore, composition/decomposition can be used. Also, it is up to the user to add further details, e.g. by using text boxes. However, there are no further abstraction concepts such as generalisation/specialisation (see F4).
A1	A modelling language should provide domain spe-	+	This requirement has been checked

	cific concepts as long as they are regularly used and their semantics is invariant within the scope of the language's application.		for all concepts of the language. In some cases, further evaluation is required.
A2	The concepts of a language should allow for modelling at a level of detail that is sufficient for all foreseeable applications. To cover further possible applications, it should provide extension mechanisms.	**o**	During requirements analysis a remarkable range of possible applications has been accounted for. However, apart from the difficulty to determine all foreseeable applications, the language currently does not allow to model position instances, which may be required by some. It does, however, include an extension mechanism ("local types").
A3	A modelling language should provide concepts that allow for clearly distinguishing different levels of abstraction within a model.	**-**	Currently, the OrgML does not allow for expressing clearly different levels of abstraction.
A4	There should be a clear mapping of the language concepts to the concepts of relevant target representations. In an ideal case, all information required by the target representations can be extracted from the model. That requires that the concepts of the language allow for expressing all concepts of relevant target representations.	**+**	Since the MEMO meta modelling language accounts for the semantics of prevalent programming languages, generating code from, e.g. for an organisational information system, a model can be based on a clear mapping. Also, marking attributes with "derivable" or "obtainable" guides a subsequent mapping to implementation languages.

Table 27: Generic Requirements for DSML

ID	Requirement		Comment
OM1	The language should include concepts to describe organisational structures and business processes on various levels of detail. It should also feature concepts that support specific analysis and design tasks (corresponds to requirement **U1**).	**o**	The level of detail can be adapted to particular modelling purposes – by adding or leaving out optional features. The range of scenarios accounted for during requirements analysis comprises various analysis and design tasks.
OM2	Requirement OM2: The modelling language	**+**	The existing technical language was

	should be aimed at reconstructing the technical language used in organisation analysis and design. This requirement is **a specialisation** of the generic **Requirement U1**.		the starting point for designing the language. In addition to that, "local types" allow for representing specific aspects of local terminologies.
OM3	The OrgML should include concepts of other modelling languages to support references to respective models. This requirement applies especially to other languages used for enterprise modelling.	+	In the given specification, only a few concepts, e.g. "Class" or "Method", are accounted for. However, further concepts can be added easily. Also, other language specifications within MEMO include concepts of the OrgML.
OM4	The OrgML should support all of those known control structures that are relevant for the purpose of the language.	+	While completeness cannot be proven, the current set of control structures is comprehensive with respect to the scope of relevant use scenarios and given catalogues (such as Van der Aalst et al. 2003).
OM5	The OrgML should allow for detailed references to models used for systems analysis and design. It should provide mappings to relevant workflow specification languages.	+	This feature has been demonstrated for a previous version of the OrgML already Jung 2004.
OM6	The OrgML should include concepts that allow for assigning model elements to different organisations/enterprises and to express relevant patterns of communication and cooperation.	+	The OrgML allows for assigning organisational units to different organisations. It also allows assigning different organisations to business processes or parts of business processes respectively.
OM7	While the specification of the language should avoid conceptual redundancy, reoccurring modelling patterns should be represented by specific model elements – if they promise to improve productivity and readability.	o	Respective concepts are not provided for modelling organisation structures. The process modelling part of the language provides various concepts of this kind, e.g. iterations, synchronisation exceptions etc.
OM8	The OrgML should include concepts that allow for specifying business process models which include information required for running simulations.	o	The concept ProtypicalPosition allows for representing positions for simulation purposes. Apart from that, simulation depends on corresponding features of further modelling lan-

			guage such as the ITML or the ResML

Table 28: General Requirements for Organisation Modelling

ID	Requirement		
SR1	To allow for elaborate analyses, it should be possible to describe organisational units in a differentiated way. This includes concepts to describe formal qualification, skills, tasks, responsibilities etc. as well as different kinds of associations between organisational units.	**+**	The language offers a wide range of respective features.
SR2	An organisational chart represents organisational units. These may be types or instances. Therefore, the OrgML should provide concepts that allow for both, defining organisational units as types and as instances.	**o**	Currently, organisational units are instances of meta types. Hence, from a formal perspective, they are types. However, often they will be treated as instances. At the same time, the language allows for assigning instance-level features, e.g. through the use of intrinsic features.
SR3	Sometimes, certain assertions do not only apply to one type of organisational unit (or role or committee), but to more. It may be, for example, that various organisational units are assigned to a certain region. To elegantly specify such commonalities, there is need for concepts that allow for expressing abstractions over a set of organisational units (or roles and committees respectively).	**o**	Categories can be used for expressing certain kinds of commonalities. However, the OrgML does not provide more elaborate concepts such as specialisation.
SR4	With the increasing spread of cross-organisational networks, joint ventures etc., it becomes more and more important to account for modelling structures that include more than one legal institution. Therefore, the OrgML should provide concepts that allow distinguishing between different organisations.	**+**	This requirement corresponds widely to OM6.
SR5	While some organisational units will usually occur only once within a particular enterprise, others – this is typically the case for positions – can	**+**	The qualification required for filling a role can be expressed by assigning the positions that may serve as a pre-

	exist in multiple instances of a certain type. Firstly, there is need to allow for differentiating between multiplicity constraints ("There must not be more than one marketing department.", "There must be one and exactly one head of the marketing department.") and actual numbers ("The current headcount of the marketing department is 26."). The notation should support a clear differentiation of these two meanings.		requisite. In addition to that, the required qualification can be described using corresponding attributes.
SR6	Assigning employees to roles can be restricted to certain constraints, e.g. to the position an employee fills or to other roles he fills or must not fill. They may also be related to specific features of an employee, e.g. his skills. The OrgML should provide concepts that allow for expressing such constraints conveniently. It is a specific challenge to account for features of employees, since they are not within the direct scope of the language.	**o**	The preconditions for filling roles can be defined to some extend by selecting position types that qualify for a role type.
SR7	There may be rules, too, that define the preconditions for joining a committee – as well as the conditions that apply to terminating a membership. They may be related to roles, organisational units or other aspects. There should be concepts that allow for expressing these rules on an appropriate level of detail (in some cases the specific complexity of corresponding regulations would exceed the scope of an organisational model).	**o**	The preconditions for joining a committee can be defined by assigning corresponding instances of PositionCategory or PotentialSuperior (i.e. instance of the concrete subtypes). Further formal constraints are not possible.
SR8	An interaction diagram should allow for representing also other types of interaction. For example, interactions could be enriched by referring to occasions, resources, tasks, subjects, communication media etc. Furthermore, the OrgML should provide concepts that allow for representing cross-organisational interactions. To adequately represent these various kinds of interactions, it will be required to make use of different kinds of diagrams/tables.	**+**	Interaction diagrams are supported. For this purpose, a range of interaction types and a corresponding notation is provided.
SR9	It should be possible to supplement the set of predefined analysis features ("critical perfor-	**-**	The current support for adding analysis features is restricted to modify-

	mance" etc.) with additional user defined features.		ing corresponding auxiliary types.
SR10	It should be possible to assign every process within a decomposition diagram to one or more business process types, it is part of.	**o**	Currently, it is possible only to assign a subprocess type to one superior composed process only. This restriction is for a serious reason: Reusing subprocess types in different contexts would require to abstract on a core of a subprocess type that is independent from its context, e.g. from the surrounding event types.
SR11	There should be specific graphical notations for business process types and decomposable processes to distinguish them from other processes.	**+**	Corresponding symbols are available.
SR12	There should be concepts that allow for describing features required for various kinds of analysis to be performed on decomposition diagrams. This includes assigning the resources that are required by every process in order to analyse potential conflicts.	**o**	Since ComposedProcess is specialised from *AnyProcess* all corresponding features are available.
SR13	There should be concepts that allow for assigning probabilities to alternative paths of executing a business process. This requirement corresponds to OM6, since it is a prerequisite for running simulations.	**+**	Probabilities can be assigned.
SR14	Many business process types are characterised through a default flow of control. In certain, rare constellations, alternative flows of control need to be activated. However, modelling all possible constellations can result in all too complex representations that are difficult to understand and that distract from the essential flow of control. Therefore, the OrgML should provide the concept of an *exception* which is used to model unusual flows of control only. This requirement corresponds to **F3**, because an exception is an abstraction. It also corresponds to requirement **OM7**, because it allows for representing reoccurring	**+**	Exceptions are supported. In addition to that, a special kind of exception, a synchronisation exception, is provided, too.

	modelling patterns in a more readable fashion.		
SR15	Business processes are usually information intensive processes. Hence, for analysing and improving their efficiency, the flow of information is of pivotal relevance. There are numerous ways how information can be transmitted through a business process. Therefore, the OrgML should provide concepts that allow for the differentiated description of the information flow.	**o**	The flow of information through a business process is widely abstracted from. Event types can be used to express certain aspects of information flow. Furthermore, assigning methods of classes to subprocess types also allows to express information flow aspects.
SR16	Business processes are pivotal for an organisation's competitiveness. Therefore it is important to evaluate and – if necessary – improve their performance. This requires concepts that allow for describing the performance of a process – e.g. by comparing its actual performance against a reference performance.	**+**	Performance indicators can be assigned on different levels of detail.
SR17	Often, it will be important to distinguish different kinds of processes, e.g. a process type that is automated from one that is only partially automated. The modelling language should provide relevant types of processes together with a self-explanatory notation. Furthermore, it should allow for defining further process types (corresponds to requirement **A2**).	**+**	The language includes various different kinds of subprocess types.
SR18	It should be possible to express process invariants (similar to class invariants known from object-oriented modelling): If, e.g., a specific resource is required for all processes of a business process, this could be expressed through an invariant; thus contributing to modelling convenience and model integrity (corresponds to requirements requirement F1 and requirement **OM7**).	**-**	There is limited support for this requirement: ObjectAllocator, PositionAllocator and RoleAllocator allow for expressing that a specific instance is required. However, except from using additional OCL constraints, there is no way to express process invariants.
SR19	There should be concepts for representing data or objects that correspond to concepts used in systems design. Thereby, friction between business process analysis and systems design could be avoided. In an ideal case, documents used for systems design such as object models could be	**+**	The OrgML allows for assigning classes and methods to subprocess types. Furthermore, it is possible to distinguish between objects of the same class.

	generated from the representation of information in business process models.		
SR20	It should be possible to represent different physical media information is stored on, such as traditional media (paper, micro fiche etc.) and various kinds of digital media. Accounting for different physical media is pivotal for analysing the efficiency of information allocation within a business process	**-**	This aspect of information logistics is widely faded out. It can be accounted for in part through specific event types.
SR21	There should be concepts that allow for expressing different levels of formal semantics, e.g. bitmap, ASCII, structured document, class. The higher the level of formal semantics, the better are the chances for processing the corresponding information automatically.	**-**	This requirement corresponds to information logistics, too, and is not covered yet.
SR 22	It should be possible to represent the information life cycle. This includes the creation, modification and deletion of information.	**o**	On the one hand, this feature is covered by corresponding event types, on the other hand, it is covered by class methods which allow specifying modifications.
SR23	Sometimes, it is important to distinguish between different instances of information objects or resources in general – or to make sure that a certain instance is used. Therefore, it should be possible to assign identifiers.	**+**	This is possible through the use of ObjectAllocator (see SR18).
SR24	In order to support the detection of media clashes, it should be possible to represent the transformation of information into a new representation. This requirement is related to requirement SR20, requirement SR21 and requirement SR22.	**-**	This aspect of information logistics is not accounted for yet. It could be addressed by assigning a respective comment.
SR25	It should be possible to **differentiate** between value and reference semantics of data that is transferred from one process to another. This is important with respect to the efficiency of a business process, since transferring values will usually be more costly than transferring references. It is also relevant for systems analysis and design. In general, reference semantics is preferably with	**-**	This advanced feature of information logistics is not covered. It is only marginally addressed by assigning methods of particular objects. Assigning a method of one object to various subprocesses indicates that reference semantics is in place.

	respect to system integrity. However, sometimes – e.g. in offline-mode – it cannot be accomplished. Transferring copies (value semantics) requires implementing specific procedures to cater for system integrity.		
SR26	The flow of information will usually include actors such as customers, suppliers or internal employees. On the one hand, it should be possible to represent the information that is requested or provided by actors. On the other hand, there should be concepts that allow for representing communication relationships between actors, e.g. cause, frequency, duration, media etc.	**-**	Not implemented yet.
SR27	To support analysing the economics of a business process, it should be possible to assign the resources that are required to execute a process.	**o**	This is partially possible by assigning organisational units, roles, software, and classes/objects.
SR28	With respect to the economics of a business process the number (or the volume) of resources is important. Therefore, it should be possible to express this aspect.	**o**	The ResML allows for expressing resource allocation which could be assigned to subprocess types.
SR29	It should be possible to specify different types of services.	**-**	Service modelling is not covered yet.
SR30	There should be concepts that allow for defining associations between services and between services and other relevant concepts such as business processes, software systems, organisational units etc.	**-**	Not implemented yet.
SR31	The OrgML should provide concepts for specifying service contracts on various levels of detail.	**-**	Not implemented yet.
SR32	The OrgML should provide concepts that allow for specifying those features of a decision scenario that are needed for analysing and improving its performance. They include quality (perceived and measured), execution time, resources (required, actually available) and associations to other decision scenarios.	**-**	Not implemented yet.
SR33	In order to describe the path, a decision is sup-	**-**	Since concepts to describe decision

	posed to take within an organisation, decision scenarios require an appropriate combination with business process models.		scenarios are not included yet, this requirements cannot be satisfied.
SR34	Different types of associations should allow for assigning organisational units, roles and committees to business process models, e.g. "in charge of", "supports", "provides technical support" etc.	**o**	It is possible to assign organisational units and positions. However, it is not possible to distinguish kinds of relationships. While a corresponding specification could be easily added, we decided to wait until there is sufficient evidence that such a feature is actually needed.
SR35	It should be possible to conveniently express constraints on instances of organisational units, roles and committees assigned to processes. If, e.g., the position "Sales Assistant" is assigned to more than one process within a business process, it may be important to express that it should always be the same position instance (i.e. the same employee) who is assigned.	**o**	Constraints on assigning instances can be expressed to some extend by using PositionAllocator.
SR36	Referring to an organisational chart maybe regarded as helpful in some cases. In many cases, it will add to a diagrams complexity and distract from the main focus, i.e. the business process diagram. Hence, the notation should allow for assigning organisational units without representing the corresponding organisational chart. It should also allow for clearly distinguishing between different kinds of assignments.	**+**	An organisational unit can be referred to by its name (presented on top of a subprocess symbol).
SR37	It should be possible to express relevant constraints on the assignment of organisational units or roles/committees to processes. For example: If organisational units and roles were assigned the tasks they are supposed to perform and all processes were decomposed into tasks, then a constraint could be applied that only those organisational units etc. can be in charge of performing a process that cover all corresponding tasks.	**o**	Subprocess types can be assigned the tasks they include. Also, position types can be assigned tasks. Therefore, checking for a corresponding match is possible. There is, however, room for a more elaborate specification of tasks, e.g. by allowing for specialisation or aggregation between task types and tasks respectively.
SR38	In addition to a set of predefined relationship	**-**	The range of predefined relationship

	types, it should be possible to define further relationship types (corresponds to requirement **A2**).		types is assumed to be sufficient. If this assumptions turns out to be wrong, a feature for adding customised relationship types will be added.
SR39	As far as possible, concepts for business process modelling should be reused. This does not only foster the maintenance of the language, it also allows for reusing associations – e.g. to organisational units, resources, classes etc. – defined for business process diagrams.	**-**	This requirement relates to concepts for modelling projects, which are not included yet.
SR40	There is need to account for concepts that are specific for project planning, e.g. to express problems, risks, challenges as well as accomplishments.	**-**	not implemented yet
SR41	With respect to the high level of abstraction to be expected for some project execution plans, they should be supplemented with guidelines for "instantiating" project instances.	**-**	not available yet
SR42	It should be possible to associate projects or project phases with concepts of other diagram types that help with analysing and (re-) designing the project categories of a firm.	**-**	not available yet
SR43	Concepts within associated models that are referred to need to be integrated into the MEMO-OrgML. This includes, for instance, concepts such as "Class" or "Operation". Furthermore, there are concepts required that allow for differentiated associations, e.g. "instantiate", "release". In order to provide for the integration both with the MEMO OML and the UML, mappings to corresponding concepts in both languages need to be defined.	**o**	Associations to "Class" or "Method" are possible. So far, instantiation and release associations are not provided. They can be realised, however, by associating corresponding methods (e.g. a method within an "Object Factory" class that allows for instantiating objects of a given class.
SR44	Concepts within the IT resource modelling language, e.g. to describe hardware, system software, networks, applications etc. need to be included in the OrgML in order to allow for defining references.	**o**	In the current specification, the interface to the ITML is limited to Software. However, it is problem to provide for a more differentiated interface. The concepts required for that are supposed to be part of the next

			version of the ITML.
SR45	Integrating OrgML models with other models used within an enterprise model requires including all concepts of the corresponding modelling languages that are used for inter-model associations.	**o**	The current specification includes some concepts of other modelling languages, e.g. “Class” or “Software”.
SR46	To cope with modifications of existing modelling languages and the creation of further languages, it might be required to provide versatile linking mechanisms on a low semantic level.	**-**	The current specification does not address this requirement.

Table 29: Specific Requirements

8 Conclusions and Future Work

This report presents an essential part of the MEMO OrgML, i.e. concepts for business process modelling. The specification has already reached a level of complexity that many will regard as too high. However, it is merely a reflection of the subject's complexity: Business processes are pivotal patterns of collaborative action in organisations. The control structures defined in the specification cover most of the so called "workflow patterns" in Van der Aalst et al. 2003) – and go beyond those in part. Nevertheless, the current specification represents an intermediate status only. This is mainly for three reasons. At first, it does not account for information logistics, i.e. the flow of information through a business process. While adding simple flow relations between subprocesses would be fairly easy to specify, such an approach would not be satisfactory. Modelling information flow in a way that allows for elaborate analysis requires specifying different media, for distinguishing between reference and value semantics and various other aspects. The resulting requirements create a remarkable additional complexity. Therefore, I decided to consider it in an additional report. Second, the evaluation of the current specification shows that a number of requirements have not been satisfied so far. Third, like any other process modelling language, the current version lacks advanced abstraction concepts. For instance, it is not possible to directly reuse a subprocess type in a further business process model – even though a relaxed conception of process inheritance provides support for reuse. Therefore, there is need to further develop the current state of business process modelling by adding abstraction concepts that foster safe and convenient reuse of model elements. Finally, the present evaluation, which is mainly based on analytically comparing requirements against language features, should be supplemented by evaluations that were gained in practical use scenarios.

References

ANDREWS, T., CURBERA, F. & DHOLAKIA, H. 2003. *Business Process Execution Language for Web Services* [Online]. Available: http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/ [Accessed Nov. 25th 2008].

BÖRGER, E. & THALHEIM, B. 2008. A Method for Verifiable and Validatable Business Process Modeling.

COALITION, W. M. 1996. Workflow Standard - Interoperability.

FRANK, U. 2006a. Evaluation of Reference Models. *In:* FETTKE, P. & LOOS, P. (eds.) *Reference Modeling for Business Systems Analysis.* Idea Group.

FRANK, U. 2011a. The MEMO Meta Modelling Language (MML) and Language Architecture. *ICB Research Report, No. 43*, University Duisburg-Essen. 2nd Edition.

FRANK, U. 2011b. MEMO Organisation Modelling Language: Requirements and Core Diagram Types. *ICB Research Report*, No. 47, University Duisburg-Essen.

FRANK, U. 2011c. MEMO Organisation Modelling Language (1): Focus on Organisational Structure . *ICB Research Report*, No. 48, University Duisburg-Essen.

FRANK, U. 2010. Outline of a Method for Designing Domain-Specific Modelling Languages. *ICB Research Report No. 42.* University Duisburg-Essen

FRANK, U. 2011d. Some Guidelines for the Conception of Domain-Specific Modelling Languages. *In:* NÜTTGENS, M., THOMAS, O. & WEBER, B. (eds.) *Enterprise Modelling and Information Systems Architectures.* Hamburg.

FRANK, U. 2012. Specialisation in Business Process Modelling: Motivation, Approaches and Limitations. *ICB Research Report, No. 51.* University Duisburg-Essen.

FRANK, U. 2006b. Towards a Pluralistic Conception of Research Methods in Information Systems Research. *ICB Research Reports, No. 7*, University Duisburg-Essen.

JUNG, J. 2004. Mapping of Business Process Models to Workflow Schemata - An Example Using MEMO-OrgML and XPDL. *Arbeitsberichte des Instituts für Wirtschafts- und Verwaltungsinformatik.* Koblenz, Germany: Universität Koblenz-Landau.

OMG. 2008. Business Process Modeling Notation. Available: http://www.omg.org/docs/formal/08-01-17.pdf [Accessed Nov. 25th 2008].

OMG. 2009. OMG: Business Process Model and Notation (BPMN). Available: http://www.omg.org/spec/BPMN/2.0 [Accessed 2011-11-23].

RITTGEN, P. 2000. Paving the Road to Business Process Automation. *European Conference on Information Systems (ECIS).* Vienna.

RUSSEL, N., HOFSTEDE, A. H. M., VON DER AALST, W. M. P. & MULYAR, N. 2006. Workflow Control-Flow Patterns: A Revised View. . *BPM-Report BPM-06-22.*

SCHEER, A. W. 2001. *ARIS - Modellierungsmethoden, Metamodelle, Anwendungen,* Berlin, Springer.

VAN DER AALST, W. M. P. 2002. Making Work Flow: On the Application of Petri Nets to Business Process Management. *Lecture Notes in Computer Science,* 2360.

VAN DER AALST, W. M. P., DESEL, J. & E., K. 2002. On the Semantics of EPCs: A Vicious Circle. *In:* M. NÜTTGENS, M. & RUMP, F. J. (eds.) *Proceedings of the EPK 2002: Business Process Management using EPCs.* Trier.

VAN DER AALST, W. M. P., TER HOFSTEDE, A. H., KIPUSZEWSKI, B. & BARROS, A. P. 2003. Workflow Patterns. *Distributed and Parallel Databases,* 14**,** 5-51.

WEHLER, J. & LANGNER, P. 1998. Petri Net Based Certification of Event-Driven Process Chains. *Application and Theory of Petri Nets***,** 286-305.

Previously published ICB - Research Reports

2011

No 48 (December 2011)

Frank, Ulrich: "MEMO Organisation Modelling Language (1): Focus on Organisational Structure"

No 47 (December 2011)

Frank, Ulrich: "MEMO Organisation Modelling Language: Requirements and Core Diagram Types"

No 46 (December 2011)

Frank, Ulrich: "Multi-Perspective Enterprise Modelling: Background and Terminological Foundation"

No 45 (November 2011)

Frank, Ulrich; Strecker, Stefan; Heise, David; Kattenstroth, Heiko; Schauer, Carola: "Leitfaden zur Erstellung wissenschaftlicher Arbeiten in der Wirtschaftsinformatik"

No 44 (September 2010)

Berenbach, Brian; Daneva, Maya; Dörr, Jörg; Frickler, Samuel; Gervasi, Vincenzo; Glinz, Martin; Herrmann, Andrea; Krams, Benedikt; Madhavji, Nazim H.; Paech, Barbara; Schockert, Sixten; Seyff, Norbert (Eds): "17th International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2011). Proceedings of the REFSQ 2011 Workshops REEW, EPICAL and RePriCo, the REFSQ 2011 Empirical Track (Empirical Live Experiment and Empirical Research Fair), and the REFSQ 2011 Doctoral Symposium"

No 43 (February 2011)

Frank, Ulrich: "The MEMO Meta Modelling Language (MML) and Lnguage Architecture – 2nd Edition"

2010

No 42 (December 2010)

Frank, Ulrich: "Outline of a Method for Designing Domain-Specific Modelling Languages"

No 41 (December 2010)

Adelsberger,Heimo; Drechsler, Andreas (Eds): "Ausgewählte Aspekte des Cloud-Computing aus einer IT-Management-Perspektive – Cloud Governance, Cloud Security und Einsatz von Cloud Computing in jungen Unternehmen"

No 40 (October 2010)

Bürsner, Simone; Dörr, Jörg; Gehlert, Andreas; Herrmann, Andrea; Herzwurm, Georg; Janzen, Dirk; Merten, Thorsten; Pietsch, Wolfram; Schmid, Klaus; Schneider, Kurt; Thurimella, Anil Kumar (Eds): "16th International Working Conference on Requirements Engineering: Foundation for Software Quality. Proceedings oft he Workshops CreaRE, PLREQ, RePriCo and RESC"

No 39 (May 2010)
Strecker, Stefan; Heise, David; Frank, Ulrich: "Entwurf einer Mentoring-Konzeption für den Studiengang M.Sc. Wirtschaftsinformatik an der Fakultät für Wirtschaftswissenschaften der Universität Duisburg-Essen"

No 38 (February 2010)
Schauer, Carola: "Wie praxisorientiert ist die Wirtschaftsinformatik? Einschätzungen von CIOs und WI-Professoren"

No 37 (January 2010)
Benavides, David; Batory, Don; Grunbacher, Paul (Eds.): "Fourth International Workshop on Variability Modelling of Software-intensive Systems"

2009

No 36 (December 2009)
Strecker, Stefan: "Ein Kommentar zur Diskussion um Begriff und Verständnis der IT-Governance - Anregungen zu einer kritischen Reflexion"

No 35 (August 2009)
Rüngeler, Irene; Tüxen, Michael; Rathgeb, Erwin P.:"Considerations on Handling Link Errors in STCP"

No 34 (June 2009)
Karastoyanova, Dimka; Kazhamiakan, Raman; Metzger, Andreas; Pistore, Marco (Eds.): "Workshop on Service Monitoring, Adaption and Beyond"

No 33 (May 2009)
Adelsberger,Heimo; Drechsler , Andreas; Bruckmann, Tobias; Kalvelage, Peter; Kinne, Sophia; Pellinger, Jan; Rosenberger, Marcel; Trepper, Tobias: „Einsatz von Social Software in Unternehmen – Studie über Umfang und Zweck der Nutzung"

No 32 (April 2009)
Barth, Manfred; Gadatsch, Andreas; Kütz, Martin; Rüding, Otto; Schauer, Hanno; Strecker, Stefan: „Leitbild IT-Controller/-in – Beitrag der Fachgruppe IT-Controlling der Gesellschaft für Informatik e. V."

No 31 (April 2009)
Frank, Ulrich; Strecker, Stefan: "Beyond ERP Systems: An Outline of Self-Referential Enterprise Systems – Requirements, Conceptual Foundation and Design Options"

No 30 (February 2009)
Schauer, Hanno; Wolff, Frank: „Kriterien guter Wissensarbeit – Ein Vorschlag aus dem Blickwinkel der Wissenschaftstheorie (Langfassung)"

No 29 (January 2009)
Benavides, David; Metzger, Andreas; Eisenecker, Ulrich (Eds.): "Third International Workshop on Variability Modelling of Software-intensive Systems"

2008

No 28 (December 2008)
Goedicke, Michael; Striewe, Michael; Balz, Moritz: „Computer Aided Assessments and Programming Exercises with JACK“

No 27 (December 2008)
Schauer, Carola: "Größe und Ausrichtung der Disziplin Wirtschaftsinformatik an Universitäten im deutschsprachigen Raum - Aktueller Status und Entwicklung seit 1992"

No 26 (September 2008)
Milen, Tilev; Bruno Müller-Clostermann: " CapSys: A Tool for Macroscopic Capacity Planning"

No 25 (August 2008)
Eicker, Stefan; Spies, Thorsten; Tschersich, Markus: "Einsatz von Multi-Touch beim Softwaredesign am Beispiel der CRC Card-Methode"

No 24 (August 2008)
Frank, Ulrich: "The MEMO Meta Modelling Language (MML) and Language Architecture – Revised Version"

No 23 (January 2008)
Sprenger, Jonas; Jung, Jürgen: "Enterprise Modelling in the Context of Manufacturing – Outline of an Approach Supporting Production Planning"

No 22 (January 2008)
Heymans, Patrick; Kang, Kyo-Chul; Metzger, Andreas, Pohl, Klaus (Eds.): "Second International Workshop on Variability Modelling of Software-intensive Systems"

2007

No 21 (September 2007)
Eicker, Stefan; Annett Nagel; Peter M. Schuler: "Flexibilität im Geschäftsprozess-management-Kreislauf"

No 20 (August 2007)
Blau, Holger; Eicker, Stefan; Spies, Thorsten: "Reifegradüberwachung von Software"

No 19 (June 2007)
Schauer, Carola: "Relevance and Success of IS Teaching and Research: An Analysis of the ‚Relevance Debate'

No 18 (May 2007)
Schauer, Carola: "Rekonstruktion der historischen Entwicklung der Wirtschaftsinformatik: Schritte der Institutionalisierung, Diskussion zum Status, Rahmenempfehlungen für die Lehre"

No 17 (May 2007)
Schauer, Carola; Schmeing, Tobias: "Development of IS Teaching in North-America: An Analysis of Model Curricula"

No 16 (May 2007)
Müller-Clostermann, Bruno; Tilev, Milen: "Using G/G/m-Models for Multi-Server and Mainframe Capacity Planning"

No 15 (April 2007)
Heise, David; Schauer, Carola; Strecker, Stefan: "Informationsquellen für IT-Professionals – Analyse und Bewertung der Fachpresse aus Sicht der Wirtschaftsinformatik"

No 14 (March 2007)
Eicker, Stefan; Hegmanns, Christian; Malich, Stefan: "Auswahl von Bewertungsmethoden für Softwarearchitekturen"

No 13 (February 2007)
Eicker, Stefan; Spies, Thorsten; Kahl, Christian: "Softwarevisualisierung im Kontext serviceorientierter Architekturen"

No 12 (February 2007)
Brenner, Freimut: "Cumulative Measures of Absorbing Joint Markov Chains and an Application to Markovian Process Algebras"

No 11 (February 2007)
Kirchner, Lutz: "Entwurf einer Modellierungssprache zur Unterstützung der Aufgaben des IT-Managements – Grundlagen, Anforderungen und Metamodell"

No 10 (February 2007)
Schauer, Carola; Strecker, Stefan: "Vergleichende Literaturstudie aktueller einführender Lehrbücher der Wirtschaftsinformatik: Bezugsrahmen und Auswertung"

No 9 (February 2007)
Strecker, Stefan; Kuckertz, Andreas; Pawlowski, Jan M.: "Überlegungen zur Qualifizierung des wissenschaftlichen Nachwuchses: Ein Diskussionsbeitrag zur (kumulativen) Habilitation"

No 8 (February 2007)
Frank, Ulrich; Strecker, Stefan; Koch, Stefan: "Open Model - Ein Vorschlag für ein Forschungsprogramm der Wirtschaftsinformatik (Langfassung)"

2006

No 7 (December 2006)
Frank, Ulrich: "Towards a Pluralistic Conception of Research Methods in Information Systems Research"

No 6 (April 2006)
Frank, Ulrich: "Evaluation von Forschung und Lehre an Universitäten – Ein Diskussionsbeitrag"

No 5 (April 2006)
Jung, Jürgen: "Supply Chains in the Context of Resource Modelling"

No 4 (February 2006)
Lange, Carola: "Development and status of the Information Systems / Wirtschaftsinformatik discipline: An interpretive evaluation of interviews with renowned researchers, Part III – Results Wirtschaftsinformatik Discipline"

2005

No 3 (December 2005)
Lange, Carola: "Development and status of the Information Systems / Wirtschaftsinformatik discipline: An interpretive evaluation of interviews with renowned researchers, Part II – Results Information Systems Discipline"

No 2 (December 2005)
Lange, Carola: "Development and status of the Information Systems / Wirtschaftsinformatik discipline: An interpretive evaluation of interviews with renowned researchers, Part I – Research Objectives and Method"

No 1 (August 2005)
Lange, Carola: „Ein Bezugsrahmen zur Beschreibung von Forschungsgegenständen und -methoden in Wirtschaftsinformatik und Information Systems"

Research Group	Core Research Topics
Prof. Dr. H. H. Adelsberger Information Systems for Production and Operations Management	E-Learning, Knowledge Management, Skill-Management, Simulation, Artificial Intelligence
Prof. Dr. P. Chamoni MIS and Management Science / Operations Research	Information Systems and Operations Research, Business Intelligence, Data Warehousing
Prof. Dr. F.-D. Dorloff Procurement, Logistics and Information Management	E-Business, E-Procurement, E-Government
Prof. Dr. K. Echtle Dependability of Computing Systems	Dependability of Computing Systems
Prof. Dr. S. Eicker Information Systems and Software Engineering	Process Models, Software-Architectures
Prof. Dr. U. Frank Information Systems and Enterprise Modelling	Enterprise Modelling, Enterprise Application Integration, IT Management, Knowledge Management
Prof. Dr. M. Goedicke Specification of Software Systems	Distributed Systems, Software Components, CSCW
Prof. Dr. V. Gruhn Software Engineering	Design of Software Processes, Software Architecture, Usability, Mobile Applications, Component-based and Generative Software Development
PD Dr. C. Klüver Computer Based Analysis of Social Complexity	Soft Computing, Modeling of Social, Cognitive, and Economic Processes, Development of Algorithms
Prof. Dr. T. Kollmann E-Business and E-Entrepreneurship	E-Business and Information Management, E-Entrepreneurship/E-Venture, Virtual Marketplaces and Mobile Commerce, Online-Marketing
Prof. Dr. B. Müller-Clostermann Systems Modelling	Performance Evaluation of Computer and Communication Systems, Modelling and Simulation
Prof. Dr. K. Pohl Software Systems Engineering	Requirements Engineering, Software Quality Assurance, Software-Architectures, Evaluation of COTS/Open Source-Components
Prof. Dr.-Ing. E. Rathgeb Computer Networking Technology	Computer Networking Technology
Prof. Dr. E. Rukzio Mobile Mensch Computer Interaktion mit Software Services	Novel Interaction Technologies, Personal Projectors, Pervasive User Interfaces, Ubiquitous Computing
Prof. Dr. R. Unland Data Management Systems and Knowledge Representation	Data Management, Artificial Intelligence, Software Engineering, Internet Based Teaching
Prof. Dr. S. Zelewski Institute of Production and Industrial Information Management	Industrial Business Processes, Innovation Management, Information Management, Economic Analyses

For more information visit us on the Web: **http://www.icb.uni-due.de**

ISSN 1860-2770 (Print)
ISSN 1866-5101 (Online)