



Technisch-Naturwissenschaftliche  
Fakultät

# **Behavior approached decomposition of the BPMN 2.0 with enhanced refinements preserving the semantics of the ground model**

## **DISSERTATION**

zur Erlangung des akademischen Grades

## **Doktor**

im Doktoratsstudium der

## **TECHNISCHEN WISSENSCHAFTEN**

Eingereicht von:

**Jan Kubovy**

Angefertigt am:

**Institut für Anwendungsorientierte Wissensverarbeitung (FAW)**

Beurteilung:

**A.Univ.-Prof. Dr. Josef Küng**

**Prof. Dr. Bernhard Thalheim**

**Linz, September, 2014**



### **Sworn Declaration**

I hereby declare under oath that the submitted doctoral dissertation has been written solely by me without any outside assistance, information other than provided sources or aids have not been used and those used have been fully documented. The dissertation here present is identical to the electronically transmitted text document.

### **Eidesstattliche Erklärung**

Ich erkläre an Eides statt, dass ich die vorliegende Dissertation selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe. Die vorliegende Dissertation ist mit dem elektronisch übermittelten Textdokument identisch.

Linz, September, 2014



Jan Kubovy



Behavior approached  
decomposition of the BPMN 2.0  
with enhanced refinements  
preserving the semantics of the  
ground model

© 2014 Jan Kubovy. All rights reserved.

*To my mother.*

*To my grandmother.*

*To my grandfather, in memoriam.*





*“The world is a dangerous place to live; not because of the people who are evil, but because of the people who don’t do anything about it.”*

— Albert Einstein



# Contents

<b>List of Figures</b>	<b>xv</b>
------------------------	-----------

<b>List of Tables</b>	<b>xvii</b>
-----------------------	-------------

<b>1 Introduction</b>	<b>1</b>
1.1 Related work and relation to the surroundings . . . . .	3
1.2 Process modeling . . . . .	7
1.3 Executable models . . . . .	8
1.4 Formal descriptions . . . . .	8
1.5 Business Process Model and Notation . . . . .	10

<b>I Abstract State Machines</b>	<b>13</b>
----------------------------------	-----------

<b>2 Introduction into Abstract State Machines</b>	<b>15</b>
2.1 Notation and conventions . . . . .	17
2.1.1 Universes and types . . . . .	18
2.1.2 Basic functions . . . . .	19
2.1.3 Derived functions . . . . .	20
2.1.4 Rules . . . . .	20
2.1.5 Auxiliary constructs, statements and keywords . . . . .	21

<b>3 The refinement method</b>	<b>23</b>
3.1 Business level of abstraction . . . . .	24
3.1.1 Business level document structure . . . . .	24
3.1.2 Using formal elements in an informal text . . . . .	25
3.1.3 Defining universes . . . . .	25
3.2 Technical level of abstraction . . . . .	26
3.2.1 Technical level document structure . . . . .	27
3.2.2 Selectors and properties of universes . . . . .	27
3.3 Transition between the abstraction levels . . . . .	28
3.4 Preserving consistency between the abstraction levels . . . . .	30

<b>II Business Process Model and Notation</b>	<b>31</b>
<b>4 Modeling elements in BPMN</b>	<b>33</b>
4.1 Flow elements . . . . .	34
4.2 Activities . . . . .	36
4.2.1 Activity flow node . . . . .	36
4.2.2 Activity control flow . . . . .	37
4.2.3 Tasks . . . . .	39
4.3 Events . . . . .	41
4.3.1 Events flow node . . . . .	41
4.3.2 Events control flow . . . . .	44
4.3.2.1 Start Events . . . . .	44
4.3.2.2 Intermediate events . . . . .	45
4.3.2.3 End events . . . . .	45
4.3.3 Triggers . . . . .	46
4.4 Gateways . . . . .	48
4.4.1 Gateways flow node . . . . .	50
4.4.2 Gateways control flow . . . . .	51
4.4.3 Event-based gateway . . . . .	52
4.5 Graphical representation . . . . .	53
<b>5 Enhanced Business Process Model and Notation</b>	<b>55</b>
5.1 Universes and types . . . . .	55
5.1.1 Activity related universes . . . . .	57
5.1.2 Events related universes . . . . .	57
5.1.3 Trigger related universes . . . . .	59
5.1.4 Gateway related universes . . . . .	60
5.2 Meta-model . . . . .	61
5.2.1 Tasks . . . . .	61
5.2.2 Gateways . . . . .	62
5.2.3 Events, triggers, notifications . . . . .	64
<b>6 Formalization of behaviors</b>	<b>69</b>
6.1 Enabling token sets . . . . .	70
6.2 Gate behavior . . . . .	71
6.3 Merge behavior . . . . .	72
6.4 Split behavior . . . . .	74
6.5 Gateway transitions . . . . .	75
6.6 Sole exclusive gateway . . . . .	76
6.7 Event behavior . . . . .	77
<b>7 The ground model</b>	<b>79</b>
7.1 General functions and rules . . . . .	81
7.2 Behaviors . . . . .	83
7.2.1 Gate behavior . . . . .	83
7.2.2 Activation behavior . . . . .	83

7.2.3	Output behavior . . . . .	89
7.3	Flow nodes . . . . .	92
7.4	Activities . . . . .	94
7.5	Events . . . . .	98
7.6	Gateways . . . . .	104
7.7	Removed parts from the original BPMN specification . . . . .	108
<b>III</b>	<b>Workflow interpreter of the workflow engine</b>	<b>111</b>
<b>8</b>	<b>Targeting the workflow interpreter</b>	<b>113</b>
8.1	Contexts . . . . .	114
8.2	Notifications . . . . .	115
8.3	Implicit throw events . . . . .	116
8.3.1	Message and signal pools . . . . .	116
8.3.2	Publication resolution . . . . .	117
8.3.3	Propagation resolution . . . . .	118
8.4	Deployments and deployment manager . . . . .	120
8.5	Upstream token . . . . .	121
8.5.1	Work of the algorithm . . . . .	122
8.5.2	Enableness test . . . . .	124
8.5.3	Cyclic workflow graphs . . . . .	124
8.5.4	Well structured processes . . . . .	125
8.5.5	Non-separable processes . . . . .	126
8.6	Transiting to the technical level ground model . . . . .	126
<b>9</b>	<b>Conclusion</b>	<b>131</b>
9.1	Results . . . . .	131
9.2	Future Work . . . . .	135
	<b>Bibliography</b>	<b>137</b>
	<b>Index</b>	<b>147</b>
	<b>Acronyms</b>	<b>151</b>
	<b>Glossary</b>	<b>153</b>
<b>A</b>	<b>Transition from business level</b>	<b>169</b>
<b>B</b>	<b>The complete BPMN ground model in a nutshell</b>	<b>175</b>
B.1	General functions and rules . . . . .	175
B.2	Token related functions and rules . . . . .	176
B.3	Behaviors . . . . .	180
B.3.1	Gate behavior . . . . .	180
B.3.2	Activation behavior . . . . .	181

B.3.3	Output behavior . . . . .	187
B.4	Flow nodes . . . . .	189
B.5	Activities . . . . .	191
B.6	Events . . . . .	194
B.7	Gateways . . . . .	201
B.8	Notifications . . . . .	204
B.9	Context . . . . .	208
B.10	Workflow Interpreter . . . . .	209
<b>C</b>	<b>ASM ground model LaTeX package</b>	<b>213</b>
C.1	ASM code . . . . .	213
C.1.1	Directory structure . . . . .	213
C.1.2	File naming conventions . . . . .	214
C.1.3	ASM file format conventions . . . . .	214
C.1.3.1	Signatures . . . . .	215
C.1.3.2	Implementation . . . . .	217
C.1.3.3	Indentation . . . . .	218
C.1.3.4	Line length constraints . . . . .	218
C.2	LaTeX documents . . . . .	218
C.2.1	Configuring the ASMGGM package . . . . .	218
C.2.2	LaTeX command provided by the ASMGGM package . . . . .	219
C.2.3	Including ASM code to LaTeX documents . . . . .	220
C.2.3.1	Description of functions and rules in LaTeX documents	223
C.2.3.2	Signatures of functions and rules in LaTeX documents	224
C.2.3.3	Implementations of functions and rules in a LaTeX document . . . . .	225
C.2.3.4	The whole function or rule in LaTeX documents . . . . .	229
<b>D</b>	<b>Curriculum vitae</b>	<b>231</b>

# List of Figures

1.1	Parts dependency graph . . . . .	2
1.2	Contributions . . . . .	6
1.3	A BPMN activity composed as a container using two simple gateways and one simple activity . . . . .	11
1.4	Parallel gateway decomposition using behavioral building blocks . . . .	12
1.5	Activity decomposition using behavioral building blocks . . . . .	12
4.1	FlowElement class and its immediate relevant children SequenceFlow and FlowNode . . . . .	34
4.2	Conditional sequence flow . . . . .	35
4.3	FlowNode class and its immediate relevant children Activity, Event, and Gateway . . . . .	36
4.4	Merging extended modeling element . . . . .	37
4.5	Merging using gateways explicitly . . . . .	37
4.6	Fork extended modeling element . . . . .	38
4.7	Splitting using gateways explicitly . . . . .	39
4.8	Tasks meta-model class based on behavioral decomposition . . . . .	40
4.9	Decomposition of events based on their behavior . . . . .	42
4.10	Event meta-model class based on behavioral decomposition . . . . .	43
4.11	Outgoing sequence flows from start event . . . . .	44
4.12	Incoming and outgoing sequence flows from and to an intermediate event . . . . .	45
4.13	Incoming sequence flows to an end event . . . . .	45
4.14	Refactored trigger class hierarchy . . . . .	46
4.15	Refactored signal, message, error and escalation class hierarchy . . . .	47
4.16	Refactored expression class hierarchy . . . . .	48
4.17	Gateway directions . . . . .	49
4.18	Decomposition of gateways based on their behavior . . . . .	50
4.19	Gateways meta-model class based on behavioral decomposition . . . .	51
4.20	Exclusive gateway depiction . . . . .	52
4.21	Instantiating with event-based gateway vs. start events . . . . .	53
4.22	Event-based gateway class hierarchy . . . . .	53
5.1	Basic decomposition of diagram contents . . . . .	62

5.2	Receive task and send task vs. task with message event . . . . .	63
5.3	Refactored gateway class hierarchy . . . . .	63
5.4	Refined expression class hierarchy . . . . .	64
5.5	Notifications class hierarchy . . . . .	66
5.6	Refactored trigger class hierarchy removing links between triggers and notifications . . . . .	67
6.1	Enabling token sets . . . . .	71
6.2	Gate behavior symbol . . . . .	72
6.3	Merge behaviors . . . . .	73
6.4	Split behavior . . . . .	75
8.1	A symmetric vicious circle . . . . .	125
8.2	A vicious circle in well structured process . . . . .	125
8.3	A non-separable process . . . . .	126
C.1	Directory Structure . . . . .	214
C.2	Documentation Example . . . . .	216
C.3	Signature Examples . . . . .	217
C.4	Implementation Examples . . . . .	218
C.5	A rule example . . . . .	221
C.6	A monitored function example . . . . .	222
C.7	Description of ExampleRule . . . . .	223
C.8	Description of the default functionExample . . . . .	223
C.9	Description of functionExample : U_A X U_B -> U_R . . . . .	224
C.10	Inline signature of functionExample . . . . .	225
C.11	Implementation of RuleExample . . . . .	228
C.12	Implementation of RuleExample with interspersed comments . . . . .	228
C.13	Implementation of the default functionExample . . . . .	228
C.14	Implementation of functionExample : U_A X U_B -> U_R . . . . .	229
C.15	Usage example of \asm command . . . . .	229



# List of Tables

2.1	Basic function visualization . . . . .	17
6.1	Enabling token sets . . . . .	71
C.1	Translation table for ASM code . . . . .	227



# Abstract

This thesis shows a decomposition of the Business Process Model and Notation (BPMN) meta-model [1] based on behaviors. The different BPMN flow node types are then composed using these behaviors defined in the BPMN 2.0 standard. Composition of the different flow node types using these behavior patterns is demonstrated with modifications of the BPMN meta-model hierarchy to improve the target Abstract State Machine (ASM) ground model [2].

A formal specification of the control flow in a BPMN Workflow Engine (WFE) is then introduced using the ASM method. We call the part of BPMN WFE, which is responsible only for the control flow of a BPMN process the Workflow Interpreter (WFI). The ASM method allows describing a system formally on a certain abstraction level [2, 3, 4, 5]. However this abstraction level can change during the process of designing a system as a result of refinements. The way a system needs to be described on an abstract level usually differs from the way a system is described when more concrete specifics are known. In this thesis we introduce two basic abstraction levels, their semantics, show how to realize transition of the related documents describing the system from one abstraction level to another and how to preserve consistency of the specification documents between both levels in case of changes. An example is made on the BPMN specification, which is first described on a business level and further transits to a technical level.

Targeting the WFI, focus is also put on simplification and reusability improvement of the resulting implementation contributing to meta-model hierarchy change. Furthermore, the BPMN non-trivial activation concept, event flow and some other parts of the BPMN specification are refined. One of core refinement is the concrete formal definition of upstream token concept and calculation of the enableness of an inclusive gateways. The introduced algorithm for upstream token calculation considers also situations where two or more gateways are mutually dependent. Another is clarification and clear separation of events, triggers and notifications, definition of message and signal pools, and formal definition of the different event forwarding concepts [1].

**Keywords:** abstraction levels, asm, behavior, bpmn, decomposition, formal methods, refinement



# Kurzfassung

Diese Arbeit zeigt eine Verhaltensweisen-basierte Zerlegung des Business Process Model and Notation (BPMN) Meta-Modells [1]. Die verschiedenen BPMN Flow Node-Typen werden dann mit Hilfe dieser Verhaltensweisen zusammengesetzt, z.B., Parallel-, Exklusive-, Inklusive-, Komplex-, Zusammenführend-, Aufspaltend-, Event-Verhalten die im BPMN 2.0-Standard definiert sind. Beispiele unterschiedlicher Zusammensetzung der Flow Node-Typen mit diesen Verhaltensmuster sind weiterhin demonstriert. Wir schlagen eine Modifikation der BPMN Meta-Modell-Hierarchie vor um des Ziel-Abstract State Machine (ASM) ground model [2] zu verbessern.

Eine formale Spezifikation des Kontrollflusses in einer BPMN Workflow Engine (WFE) wird dann eingeführt mit Anwendung der ASM-Methode. Wir nennen den Teil von BPMN WFE, der nur für den Kontrollfluss eines BPMN Prozesses verantwortlich ist, Workflow Interpreter (WFI). Die ASM-Methode ermöglicht ein System formal in einer bestimmten Abstraktionsebene zu beschreiben [2, 3, 4, 5]. Allerdings kann sich diese Abstraktionsebene während des Prozesses der Entwicklung eines Systems als Folge von Verbesserungen verändern. Die Art, wie ein System auf einer abstrakten Ebene beschrieben ist unterscheidet sich, üblicherweise von der Art, wie ein System beschrieben ist, wenn konkretere Einzelheiten bekannt sind. In dieser Arbeit stellen wir zwei Grundabstraktionsebenen vor, ihre Semantik, zeigen, wie man den Übergang der zugehörigen Dokumente, die das System beschreiben, von einer Abstraktionsebene zur anderen realisieren kann und wie im Falle von Änderungen die Konsistenz der Spezifikations Dokumente zwischen beiden Ebenen erhalten werden kann. Zwei Hauptabstraktionsebenen werden vorgestellt, um eine mögliche Klassifizierung von Abstraktionsebenen einer formalen Spezifikation zu zeigen. Darüber hinaus sind die grundlegenden Unterschiede und die Richtlinien für den Übergang zwischen diesen Ebenen diskutiert. Ein Beispiel ist an der BPMN-Spezifikation gemacht, die zuerst auf einer Geschäftsebene beschrieben ist und dann zu eine technischer Ebene übergeht.

Bei der WFI kommt die Vereinfachung und Verbesserung der Wiederverwendbarkeit der entstehenden Implementierung in den Fokus, die auch zu Änderungen von der Meta-Modell-Hierarchie beiträgt. Weiterhin sind das nicht-triviale Aktivierungskonzept, Event-Fluss und einige andere Teile der BPMN Spezifikation verfeinert. Eine der Kern Verfeinerungen ist die konkrete formale Definition des upstream token-Konzepts und die Berechnung des Enabledness eines Inklusive Gate-

ways. Der eingeführte Algorithmus zur upstream token Berechnung berücksichtigt auch Situationen, in denen zwei oder mehr gateways gegenseitig Abhängig sind. Eine weitere Verfeinerung ist unter Anderem die Klärung und Trennung von events, triggers und notifications, Definition von message und signal Pools und formale Definition der verschiedenen event Weiterleitungskonzepte [1].

## Acknowledgements

I take this opportunity to express my profound gratitude and deep regards to my guide A.Univ.-Prof. Dr. Josef Küng from the Institute for Application Oriented Knowledge Processing at the Johannes Kepler University in Linz<sup>1</sup> for his exemplary guidance and support throughout the course of this thesis. I would like to express my deep sense of gratitude to Prof. Dr. Egon Börger from Dipartimento di Informatica Università di Pisa<sup>2</sup>. My knowledge is still very less to express the sincere thanks to him for his valuable time he devoted to consult me and for the many helpful suggestions provided which found to be very helpful in construction of the thesis.

I also take this opportunity to express a deep sense of gratitude to Dr. Mag. Dagmar Auer also from the Institute for Application Oriented Knowledge Processing at the Johannes Kepler University in Linz for her constant encouragement, help, stimulating discussions, and time she devoted for reviewing this thesis and other publications during the period of my assignment.

Besides my advisers, I would like to thank Univ.-Prof. Roland Wagner from the Institute for Application Oriented Knowledge Processing at the Johannes Kepler University in Linz and Gabriela Wagner from DEXA Society<sup>3</sup> with whom I had the chance to participate on the organization of an international conference.

I am obliged to staff members especially of FAW GmbH<sup>4</sup> and SCCH GmbH<sup>5</sup>, for the valuable information provided by them in their respective fields. I am grateful for their cooperation during the period of my assignment.

I would like to thank many people who have taught and motivated me for their kind assistance, giving wise advices and helping with various applications.

Last but not the least, I would like to thank my family, especially my mother a for giving birth to me at the first place, raising me and supporting me both spiritually and financially throughout my studies and the whole life. And to my grandparents for their constant encouragement without which this assignment would not be possible.

---

<sup>1</sup><http://www.faw.jku.at>

<sup>2</sup><http://www.di.unipi.it/>

<sup>3</sup><http://dexa.org>

<sup>4</sup><http://www.faw.at>

<sup>5</sup><http://scch.at/>





---

This work was supported in part by the Austrian Science Fund (FWF) under grant no. TRP 223-N23.

---



*As practiced by computer science, the study of programming is an unholy mixture of mathematics, literary criticism, and folklore.*

— Beau Sheil

# Chapter 1

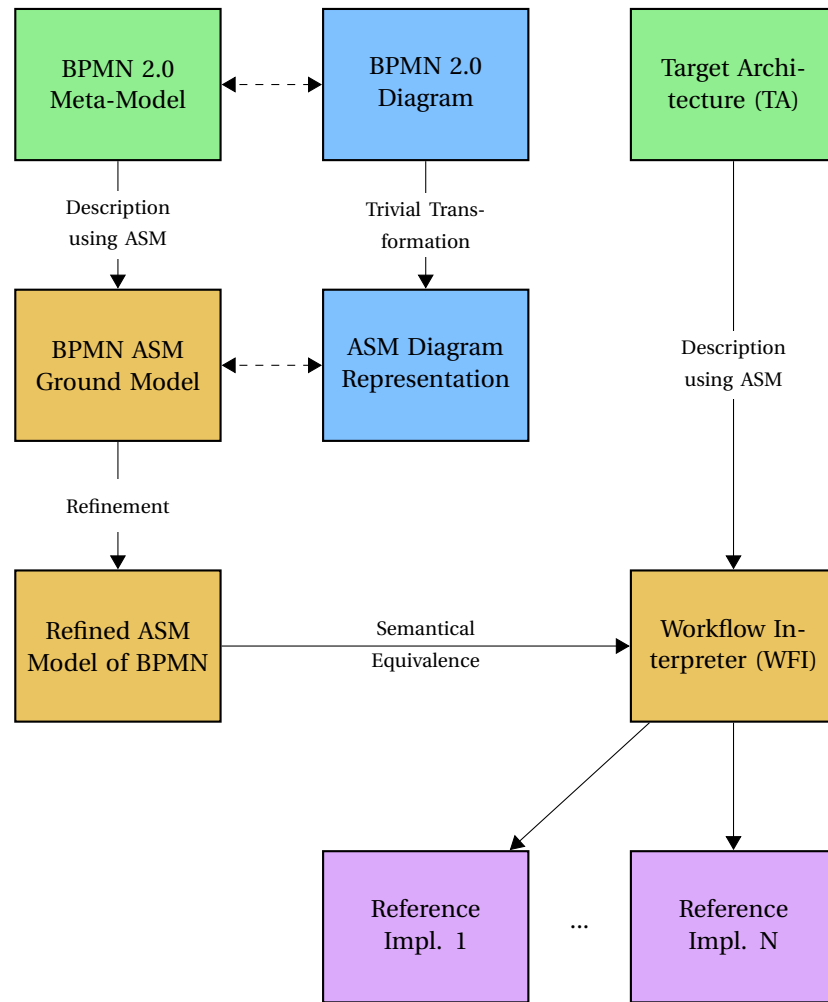
## Introduction

The goal of this thesis is formalization of the Business Process Model and Notation (BPMN) [1] meta-model and preservation of its semantics during refinements targeting a Workflow Interpreter (WFI). Figure 1.1 shows a rough decomposition of the work presented in this thesis. We start with the evaluation of the BPMN 2.0 standard and its underlying meta-model enabling BPMN 2.0 Process Diagrams (PDs). In section 1.1 with related work done by others, which significantly influenced or was taken as a base of parts of this thesis. Then in section 1.2 we discuss the gross evolution and main classification of current Work Management (WM) techniques, related modeling notations and provided meta-models with their main differences and with the focus on those, which were consulted for this work. In section 1.3 we shortly discuss the main different application models of those modeling notations relevant for this thesis. Next in section 1.4 we go into formal descriptions and their relations and uses in association with the BPMN. Different approaches and formalism are discussed in this section in brief to position this thesis between related work. The given BPMN standard is then studied further with a formal ground model [2] of its meta-model. In section 1.5 we shortly introduce this notation and its meta-model, discuss its application and current issues and introduce the decomposition approach of the BPMN meta-model further made in this thesis. The goal is to make the meta-model more compact and graspable. The intended modifications will be in favor of future refinements in the vertical direction and extensible meta-model in the horizontal direction. With vertical refinements we address the process starting with the high-level meta-model, which will continuously be refined by adding more concrete specifics incrementally, resulting in an implementation. Such refinements should happen without any gap between the original high-level specification and the implementation. By horizontal extensibility we contemplate the ability of a model to be easily and consistently extended. In the specific case of this thesis the focus is to enable the introduction of additional attributes to existing elements or even enrich the modeling language with new elements while preserving the consistency of the modeling language and the underlying meta-model.

As a formalization method the Abstract State Machine (ASM) method is used. In Part I this method, with its notation and conventions further used in this thesis,

and the refinement method is introduced. In Part II the BPMN parts relevant for this thesis will be discussed in detail with the meta-model evaluation, modification, decomposition and formalization resulting into a ground model [2]. Such formal ground model is further refined in Part III in the direction of a WFI ground model to provide a common ground for future compatible implementations.

**Figure 1.1** Parts dependency graph



## 1.1 Related work and relation to the surroundings

Contributions of this thesis and relations to the surrounding work will be discussed in this section. The relations between these units can be seen in Figure 1.2. The following units can be considered:

**BPMN 2.0** specification [1] is the starting point of this work. In the time of writing this thesis BPMN 2.0 was a well known and widely used and understood standard. Especially its graphical notation was adopted by wide range of users and companies.

Even when with September 2013 the BPMN as of version 2.0.1 [6] has become an ISO/IEC-Standard (reference number 19510:2013 [7]) it still lacked a complete formal semantics. The BPMN is considered as an important starting point [8, 9, 10] nevertheless some crucial problems we found and addressed using different methods and many efforts to add a formal semantics to the existing graphical notation with its underlying meta-model were put into this topic [11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22]. Those problems challenges both the vertical refinements and the horizontal extensions to BPMN. A formal technique to model workflows has already been considered in the literature before BPMN. Formalizing existing workflow models by mapping them to well established formalism, e.g., Petri Nets (PN) [23, 24].

When BPMN 1.0 [25] emerged it became fast a popular and widely used tool mostly because of its intuitive graphical notation. Many implementations of a Workflow Engine (WFE) using BPMN were developed since then as a response, e.g., Activity [26], Bonita Execution Engine [27], or Enhydra Shark [28], JBoss jBPM [29], Route [30]. But since the underlying meta-model lacks clear semantics those implementation do not implement the BPMN standard in the same way resulting in non-interchangeable input/output formats of those tools. Since Activity Diagrams (ADs) [31] were taken as the basis for BPMN similarities between those two graphical notations can be observed, e.g., decision gateways or activities to model actions. BPMN further evolves and in version 2.0 it incorporates concepts, i.e., tokens from PN [32].

During the time BPMN is used also many best practices evolved [33, 34, 35, 36, 37], containing among others gateway pairing or restriction of sequence flows on one flow node side. Those practices target improved readability of simple scenarios, increase clearness and reduce ambiguity of even complex scenarios and identification and reduction of common mistakes in modeling sometimes due to the unclear meaning or insufficient definition of modeling elements in the BPMN standard. In realistic scenarios a diagram can easily become too large that it is virtually impossible to grasp and communicate it as a whole. One possible solution is to deal with them in chunks (containers or macros).

**ASMs** formerly evolving algebras [38, 39, 40, 41, 42, 43] described in [2] are a given formal modeling method used in this thesis. The choice of this particular method is grounded in the related project aiming to formalize the BPMN 2.0

specification [44] and Austrian Science Fund (FWF) project “Preserving Semantics during Refinement of Business Processes” under grant no. TRP 223-N23, which was the main part of the research this thesis. The ASM method has actively been used to formalize parts of the BPMN and other workflow notations in the past already [11, 16, 17], and also for extending the BPMN standard, e.g., [45, 46]. The results of this work and parts of the resulting BPMN ASM ground model is one of the basis of this thesis.

Software engineers will find this method rather intuitive since the notation appears to look like pseudo code [2], which is a commonly used form to describe algorithms before implementing them. ASMs are very flexible especially with respect to its own notation and come with a formally defined semantics [2]. Such flexibility in notation makes it possible to come up with formulations very close to natural language. The ASM method is used to describe algorithms which distinguish it from many other commonly known formal methods. The widely applied B [47, 48] and Event-B [49] methods describe algorithms similarly. The ASM method concentrates on the dynamics of a system and makes the intentions of a model easier to grasp. ASM models can also be simulated using specialized support tools, e.g., the CoreASM [50, 51, 52], allowing to validate the models even before any implementation attempt. Use of formal methods for software and system specification allows identifying ambiguities, inconsistencies, duplicates and other flaws within an initial natural language requirements specification in an early stage of the development process.

**Java and JVM Book** provides a structured and high-level, modularized description, together with a mathematical and an experimental analysis, of Java and of the Java Virtual Machine (JVM) with a refinement to executable machines [53]. This work is used as a methodology for refinements and decomposition as shown in Figure 1.1, where the BPMN corresponds to Java language and Target Architecture (TA) corresponds to JVM. This approach helps us to develop the two main parts the BPMN ground model and its vertical refinements in the direction of executable models [54, 44] separately from the TA [55], which is not part of this thesis, where most of the horizontal refinements take place [56].

**Modeling the Semantics of BPMN as an ASM Ground Model** is a formalization approach to describe BPMN 2.0 specification using the ASM method [44]. The ground model - or “*Blueprint*” - of the BPMN 2.0 specification can be seen in chapter 7. The BPMN ground model, which is a subset taken from [44] and modified for the needs of this thesis, will be further refined by the stepwise refinement method [2]. Such refinements are without any restriction to certain forms of programs or programming languages or to programs with *monolithic state operations* causing the so called *Frame Problem* [57].

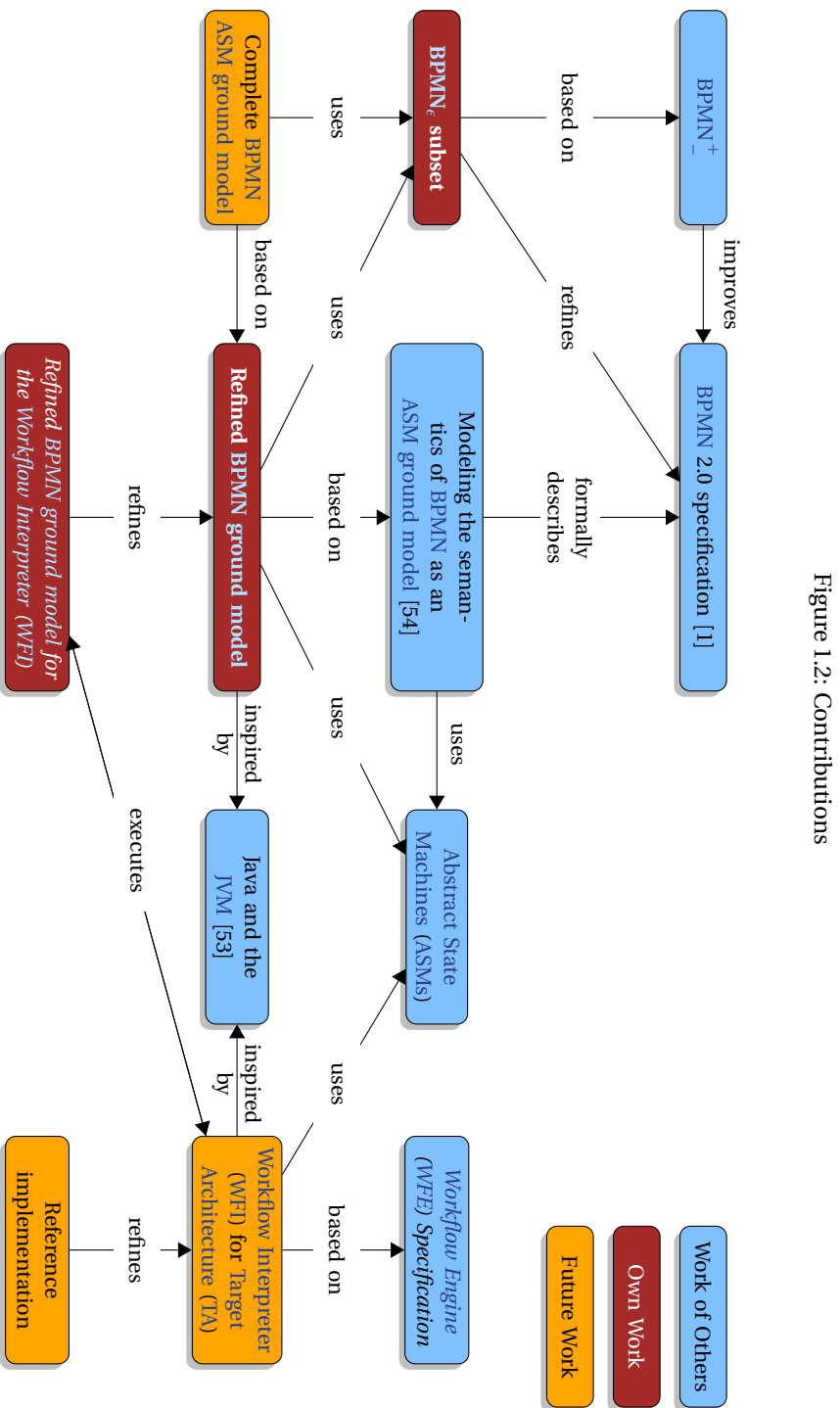
**BPMN +/-** a not yet coherent approach of identifying inconsistencies and ambiguities in the BPMN 2.0 specification [1] partially implemented in [44]. The

goal of this effort was to identify superfluous, unnecessary elements, duplications and also underspecified elements in the BPMN standard and address those with an appropriate solution proposal. Among those elements are, e.g., event-based gateways [58] and message-tasks [44].

**Enhanced Business Process Model and Notation (BPMN<sub>ε</sub>)** is a subset of the BPMN 2.0 specification cut down to the necessary but complete minimum needed for the purpose of this thesis, the control flow, the event flow and other parts from the original meta-model necessary for this thesis. The purpose of the BPMN<sub>ε</sub> is to address the collected ambiguities and inconsistencies in the BPMN 2.0 specification [1] such as superfluous, unnecessary elements, duplications and underspecified elements based on the BPMN +/- discussed above and propose corrections as well as enhancements and simplifications. The solution is based on improved meta-model decomposition. The changed meta-model should not affect the modeling scope of BPMN 2.0 specification but allow to reuse its part in different modeling elements. Allowing, e.g., duplicate definition of behaviors in different BPMN elements to be merged into one chunk reused in different places of the improved meta-model causing more error-prone specification. The resulting meta-model should still allow backward compatible graphical models with the original BPMN 2.0 specification.

**Refined BPMN Ground Model (BPMN<sub>GM</sub>)** is an ASM ground model of the BPMN<sub>ε</sub>. BPMN<sub>ε</sub> and the ground model from [17, 44] were taken as a starting point and refined in the direction of executable WFI in Part III. Before unspecified parts will be add to the ground model, e.g., contexts serving as a instance wrapper implementing an event flow missing in the original BPMN specification; notifications approaching the ambiguity in the use of the term *event* in the original BPMN specification; and definition of an algorithm computing an upstream token based on [59], compared against [60], which is reused in inclusive and complex gateway.

Upstream tokens discussed in the literature, e.g., [1, 60], is a token, which may still arrive to a flow node and in some case, e.g., inclusive or complex gateway, the flow node should wait for the arrival of such token before its activation [1, tab. 13.5]. This is defined using inhibiting and anti-inhibiting paths [60] as a token, which has an inhibiting path but no anti-inhibiting path to the corresponding flow node.





## 1.2 Process modeling

In the begin of the manufacturing sector the process paradigm begun to emerge with the goal to improve efficiency and productivity [61]. With the process approach, work is regarded as a series of activities depending on each other with some additional constrains, e.g., events, gateways. These activities are further decomposed into atomic tasks, which are performed by human actors at first and by automated artificial actors later. Usually the degree of structure detail allowed the later with the access to new technologies in the 20<sup>th</sup> century. An actor is a human or an artificial task force performing well structured work. Such work is usually highly repeatable and with a high level of specification detail.

Nowadays, developed societies are no longer primarily industrial ones. They developed towards *knowledge societies*, which has a deep impact on the work force. This results in significant increase of so called knowledge work [62, 63], i.e., work requiring or depending on a certain level of creativity, individuality and self responsibility. Such work is often seen as inconsistent or not repeated enough so that any structure can be inferred [61]. This does not necessarily mean that structuring such work is not possible [10]. In contrast to knowledge workers, actors discussed above, require low level of unstructured, creative knowledge. Knowledge is not completely absent but rather specified in detail not relying on the actor's creative and inconsistent individuality<sup>1</sup>.

The focus in this work is on well defined and structured processes. WM and orchestration is an important area in both, practice and research. In the last decades, approaches based on Ford's assembly line principle have been introduced and applied not only in production but also in office automation. This led in Workflow Management (WfM), which is now the base of the mainstream Business Process Modeling (BPM). The BPM approach aims to identify, describe and improve (performance, efficiency, costs, etc.) well structured, well defined, highly repeatable and highly predictable processes.

BPM aims to model, execute, analyze, improve, and control a business process. Modeling is seen as identification of existing processes, resources, constraints, and other properties the new process has to work with and as requirement capture for the new process to be designed. Execution can be intended as manual, semi automatic or automatic. With manual execution the process model will be used as a guideline or best practices for human workers, to lead them through the process or populate knowledge from experienced human workers to others. A semi automatic execution of a process, which is currently common type of execution [64], also deals with a certain amount of tasks automatically, e.g., by calling software services [65, 66]. Since automatic execution is not always possible for various reasons, human interaction may be necessary. Reasons may include complexity of input or output, complexity of constraints, hardly describable knowledge, or various interactions with the real world not possible for an artificial worker to perform, e.g., due to technical limitations.

---

<sup>1</sup>The definition of actor and knowledge worker may vary in the literature. We use these definitions here to clearly divide the two approaches and show the different requirement put on each of such group.

### 1.3 Executable models

There are several modeling notations on the market used to model executable processes. In this section we mention only a subset of popular and commonly used notations, relevant for the work in this thesis.

**Unified Modeling Language (UML) Activity Diagram (AD)** is one of the first commonly accepted representations of workflows, to model both, computational and organization processes. This notation allows defining work using actions, diamond shaped decisions, uncontrolled splits and joins using bars, start and end of the workflow. The evolution of this notation started with the UML state diagram resulting in UML 1.x [67, 68] as a specialized case of such state diagrams and redesigned in UML 2.x [69] with PN as the basis for this redesign.

**Business Process Model and Notation (BPMN)** is an Object Management Group (OMG) standard, which deals with business processes and their modeling. It defines the graphical notation and the underlying meta-model. Similarities between BPMN and ADs can be identified (such as decision gateways, or activities to model actions) as ADs were taken as the basis for the first version of BPMN. BPMN also evolves and is in version 2.0 in the time of writing this thesis. Significant differences can be observed between the two major versions of BPMN (1.0 and 2.0). As of September 2013, the standard was certified as an ISO/IEC standard [7] in version 2.0.1 [6].

**Business Process Execution Language (BPEL)** is an Organization for the Advancement of Structured Information Standards (OASIS) standard [70] for executing processes using Web Services (WSs). The OMG also provided mapping between BPMN and Business Process Execution Language (BPEL) included in the BPMN 2.0 specification [6].

**Case Management Model and Notation (CMMN)** is an OMG standard [71], still in the time of writing this thesis in a beta version, which deals with Case Management (CM) cases and their modeling. It defines the graphical notation and the underlying meta-model to deal with semi structured, dynamic processes. It aims also to provide an execution semantics for CM but in the time of writing this thesis this was still in an early stage. We mention Case Management Model and Notation (CMMN) here for the sake of completeness, since this new standard already (in time of writing this thesis) became an important topic as an alternative or maybe even next evolution step from BPMN [10, 64].

### 1.4 Formal descriptions

The usage of formal methods is motivated by the expectation that they contribute to the correctness, robustness, and reliability of the design. The need to use those methods increases with the growing complexity and size of systems to be developed. During the last years we can observe that systems, both software and hardware, grow

in size and complexity and such development leads to larger space for errors. As the number of potential errors increases, also the probability of unwanted consequences, e.g., higher expenses on testing and bug fixing, loss of money because of product malfunction, increased time in development because of more time expensive testing and debugging, but also putting human life in danger in case of human safety critical systems. One major goal of software engineering is to enable the construction of reliable systems regardless of their complexity [72]. And one of such attempts are formal methods used for both, specification and verification of such systems by using mathematics based techniques.

The BPMN is a well established industry standard in the area of BPM. Although BPMN is claiming a formal and clear underlying semantics, some problems and contradictions have been discovered. Those problems were addressed by the scientific society using different formal methods, such as Petri Nets (PN) [20] or the ASM method [15, 17]. Much effort has been put into providing a formal layer, enhancing the existing informal meta-model for the BPMN standard. For the formal modeling we use the ASM method [2], a formal software engineering method (see [73, 4, 74, 75, 53, 15]), which was developed from evolving algebras [38, 39, 40, 41, 42, 43]. When formalizing the BPMN meta-model, our refinements follow the process execution conformance [1, sec. 2.2], concentrating on the particular part, we call behaviors.

The need for a formal technique to model workflows has already been considered in approaches before. Formalizing existing workflow models by mapping them to a well established formalism, i.e., PN [11] further evolved into Yet Another Workflow Language (YAWL) trying to satisfy the needs of universal organizational theory and standard BPM concepts [23]. About the same time BPMN emerged and became a popular, widely used standard in the area of BPM, among others, because of its intuitive graphical notation and mapping to BPEL [70].

Some may argue that since OMG provided a mapping between BPMN and BPEL, BPMN already possesses an underlying formal semantics. We will further show that the informal part of the BPMN standard has certain flaws and that only a subset of the BPMN specification has the ability to be mapped to BPEL and run on some WFE. This may result in an inconsistent interpretation of the BPMN specification in different WFEs.

Work in this thesis is tightly related to other ongoing projects, which are concerned about BPM, formal methods and a complete formal ground model of the BPMN specification. In this work we will use the results of those projects but concentrate only on the control and event flow of the specification. The data layer and the resource layer of the BPMN standard will be left open for future work. Furthermore, we will not consider parts of the standard, which are not required for process execution conformance or for BPEL process execution conformance, since in this work we concentrate on executable models and their formal semantics. Parts such as collaboration [1, ch. 9] or choreography [1, ch. 11] explicitly state in their introductions that they are not required for neither process execution conformance nor BPEL process execution conformance.

## 1.5 Business Process Model and Notation

The Business Process Model and Notation (BPMN) in current version 2.0 [1], is well established for describing business processes. With September 2013 BPMN has become an ISO/IEC-Standard (reference number 19510:2013). While BPMN represents an important milestone [8, 9, 10], it still faces some critical problems approached by both, the scientific community and the industry in different ways. Ambiguities and inconsistencies in the standard are addressed by the scientific society using different formal methods, e.g., PN [20], or the ASM method [16, 17] among others. We focus on formally describing the BPMN standard, deal with those inconsistency problems and work on the meta-model structure to improve reusability and extensibility. Those problems appear in both, the vertical refinements and the horizontal extensions to BPMN. When formalizing the BPMN meta-model, duplicates appeared to be the reason for many of the inconsistencies in the existing standard. The need for a formal technique to model workflows has already been considered in approaches before BPMN. E.g., formalizing existing workflow models by mapping them to well established formalism discussed earlier. About the same time BPMN 1.0 [25] emerged and became a popular, widely used standard in the area of BPM, among others. Much effort has been put into providing a formal layer, enhancing the existing informal meta-model for the BPMN standard. Many implementations of a BPMN WFE are already on the market as a response to both the industry and open source communities to the popularity of the BPMN. E.g., JBoss jBPM [29], Activity [26], Bonita Execution Engine [27], Route [30], or Enhydra Shark [28]. Also because of the ambiguous underlying semantics the formats are neither fully interchangeable nor do they implement the standard in the same way.

Work in this thesis is based on PDs as described in the BPMN 2.0 specification. This particular notation is chosen since it is popular between process designers and modelers and is the most commonly used graphical representation for business processes [76] in the time of writing this thesis. We believe that the usability of our outputs will be supported by the fact that the potential reader does not have to work him/herself into a completely new process modeling notation.

Nevertheless, our refinement follows the process execution conformance [1, sec. 2.2] and the BPEL process execution conformance [1, sec. 2.3] but may introduce some new details or changes to improve extensibility and remove inconsistencies and ambiguities present in the BPMN 2.0 specification. Even though the specification [1, sec. 2.2.1] explicitly states, that:

The BPMN execution semantics have been fully formalized in this version of the specification.

We start with the BPMN modeling practices, studying best practices found in the literature [33, 34, 35, 36, 37]. Such practices contain among others gateway pairing or restriction of sequence flows on one flow node side. This may improve readability of simple diagrams, increase clearness and reduce ambiguity of even complex diagrams, but increases complexity of the diagrams in general since the amount of flow nodes in a PD dramatically bursts. By simplifying the meta-model, we achieve

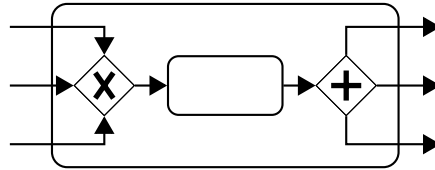
an unambiguous, clear, correct and more graspable meta-model, which is important for conformance and deployment. But the complexity growth of such process models may be harder to maintain and such meta-model may also lower the sustainability of a process model.

One possible solution is to introduce containers (for the use in graphical notations) or macros (for the use in formal descriptions). We demonstrate this, e.g., by restricting all flow nodes except gateways to only one incoming and one outgoing sequence flow. Handling the splitting and merging behavior will then remain in only one place, the gateway. Still, during the modeling of a process we do not want to place a gateway in front or after each flow node merging or splitting more sequence flows. Thus, containers may be introduced to model extended modeling elements. Such containers will then hold multiple elements from the meta-model and establish a complex element keeping the meta-model clean from problems, e.g., duplicate definitions, but enables the advantages of simple (read small, graspable and reusable) modeling elements. This way we may concentrate on the essential purpose of the different flow node types and compose complex flow elements defined in the BPMN standard using such *simple flow elements*. An example of a *BPMN Activity* as a container using a *simple activity* and two *simple gateways* is depicted in Figure 1.3. Here we define a *simple activity* limited to exactly one incoming and one

---

**Figure 1.3** A BPMN activity composed as a container using two *simple gateways* and one *simple activity*

---



outgoing sequence flow with no merging or splitting behavior. The possibility of multiple incoming or outgoing sequence flows, as described in the BPMN 2.0 standard [1, tab. 7.2], is enabled with the two gateways. The first is a *simple exclusive merging gateway*, allowing only to merge the flow using XOR-Join and the second is a *simple parallel splitting gateway* allowing only to split the flow in to parallel paths. In the BPMN 2.0 standard both, the activity and the exclusive gateway, define the exclusive merge behavior which leads to duplication in the standard and may also lead to duplication in the implementation of a WFE. Similarly for parallel split behavior of activity and parallel gateway. We identify such behaviors, which can be extracted and defined separately. We compose the complex flow elements using such behaviors. An advantage of this approach will be demonstrated by example in section 6.6 by extending the BPMN 2.0 meta-model with a *sole exclusive gateway*.

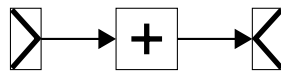
A slightly different approach from containers would be, as already sketched, to define behavior patterns [77], similarly to [17], and then reuse such patterns as building blocks to define the BPMN elements. This has the advantage that the inter-

mediate step of defining *simple flow nodes* is not necessary. We can also dynamically switch those building blocks using polymorphic or other techniques. This enables us to define one element instead of multiple elements describing all possible combinations. E.g., the activity example in Figure 1.3 would result in more than one such container if we would consider also activities with no incoming or no outgoing sequence flows or any combinations of the above. An example of a parallel gateway behavior decomposition is shown in Figure 1.4. Using such behavioral building blocks

---

**Figure 1.4** Parallel gateway decomposition using behavioral building blocks

---

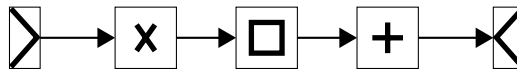


will result in Figure 1.5 to model the BPMN Activity. This time we do use much simpler building elements to compose the resulting flow node than using containers in Figure 1.3. This improves the reuse of such building elements and the resulting meta-model definition will decrease in size and complexity. This is the core decom-

---

**Figure 1.5** Activity decomposition using behavioral building blocks

---



position idea we will further use in this work to formally model the BPMN elements. This introduction should only sketch the abilities and advantages of behavioral decomposition. Furthermore, this goes hand in hand with the requirement capture [2], as we tend to describe the behavior of each modeled element and split it into small parts based on similarities and observation.

We use a square or rectangle with half of the width than height, both with sharp corners to depict a behavior. We chose this since it does not collide with any BPMN symbol. Inside this square resides the BPMN symbol identifying the basic behavior. Basic behavior means that, e.g., activity behavior does not deal with multiple incoming or outgoing sequence flows, neither does any of the gateway behavior, just the decision or work is done inside those behaviors building blocks. The quantity of the sequence flows is represented by the input or output behavior, which is shown as join and split behavior in Figure 1.4 and Figure 1.5.

*An algorithm must be seen to be believed.*  
— Donald Knuth

# Part I

## **Abstract State Machines**

In this part the Abstract State Machine (ASM) method will be discussed. The notation and syntax used further in this document and the refinement method will be defined.





*Controlling complexity is the essence of computer programming.*

— Brian Kernigan

## Chapter 2

# Introduction into Abstract State Machines

As a formal method the Abstract State Machine (ASM) method [2] was given as this method is used in projects related to this thesis, e.g., *Modeling the Semantics of Business Process Model and Notation (BPMN) Models as an ASM Ground Model* [44]. This method has already been used to for the purpose of formalization of parts of the BPMN and other workflow notations [16, 17], and also for extending the BPMN standard [45, 46]. The resulting BPMN ASM ground model is one of the foundations of this thesis.

People involved in software development will find this method rather intuitive. The notation appears to look like pseudo code [2], a commonly used form to describe algorithms before implementing them in a particular programming language. ASM also comes with a formally defined semantics while still being very flexible, especially with respect to its own notation, data structures and the level of detail in the models [2]. Such flexibility in notation makes it possible to come up with formulations very close to natural language. We will use this flexibility in this thesis to define two main abstraction levels, the business level in section 3.1 and the technical level in section 3.2. In the business level of abstraction this will allow us to create a bridge between business people, users, scientific community and developers of business support tools.

The ASM method is used to describe algorithms among others. This distinguishes it from many other commonly known formal methods. E.g, the widely applied B [47, 48] and Event-B [49] methods describe algorithms similarly. Formal specifications are often associated with statements of static constraints, such as pre- and postconditions of algorithms. Such methods have the advantage that by restricting the specification to what shall be implemented without any limitations to the implementation. They give full freedom to architects, designers and developers to find the most efficient implementation. However, specifications using only static constraints are hard to understand, not only for business people, but even for technical ones, such as developers. Most importantly, a reader cannot imme-

diately understand what is going to happen. People tend to think more in the way what they expect to happen than in terms of conditions [78]. The ASM method concentrates on the dynamics of a system and makes the intentions of a model easier to grasp. ASM models can also be simulated using specialized support tools, e.g., the CoreASM [50, 51, 52], allowing analysts and designers to validate their models even before any implementation attempt. Still ASMs remain understandable for a wide range of business people such as project owners, project managers, analysts, designers and developers. The use of formal methods for software and system specification allows identifying ambiguities, inconsistencies, duplicates and other flaws within an initial natural language requirements specification in an early stage of the development process.

ASM models are state machines (automata) describing a system, its discrete states and transitions between those states. In contrast to Finite State Machines (FSMs), single state can be described by arbitrary data structures, potentially allowing an infinite number of states. The basic concept of the ASM method described in [2] is first, the requirement capture, called ground model method, and second, advancing such a ground model by incremental refinement steps into a more concrete, possibly executable model. The ground model - or "*Blueprint*" - of the BPMN 2.0 specification can be seen in chapter 7. The BPMN ground model, which is a subset taken from [44] and modified for the needs of this thesis, will be further refined by the stepwise refinement method. Such refinements are without any restriction to certain forms of programs or programming languages or to programs with *monolithic state operations* causing the so called *Frame Problem* [57]. The key properties of the ASM method are the following [2]:

**state** can be described by arbitrary data structures and so dropping the restriction of limited state combinations [2].

**parallel execution** is the default behavior of ASM rules. All defined rules are executed in parallel in every step, and also the body of a rule is executed in parallel, i.e., all updates are carried out in parallel - if not explicitly stated otherwise, e.g., using a `seq` or a `seqblock` statement [2].

Even though, there are support tools able to execute ASM models, such as the mentioned CoreASM [51], the ASM method is meant for high-level design [2] and so comes with a performance drawback in case of executing such high-level models. To understand this we will show the basic idea behind ASMs. There are three basic building blocks: basic functions, derived functions, and rules.

**Basic functions** [2] are defining where data are stored, although they say nothing about the data, such as format or type. We can visualize basic functions as a table. The name of the function corresponds to a table name and the parameters of the function correspond to columns of such table. Substituting parameters of a basic function with concrete values will give us a concrete location and such a function call will return the value stored in the table in that location. By default every combination of any values points to a valid location. Even if no update has written into

**Table 2.1** Basic function visualization

value

value<sub>C</sub>

(a) 0-dimensional

parameter

x<sub>1</sub> = 1

x<sub>1</sub> = 2

(b) 2-dimensional

x<sub>2</sub> = 1

x<sub>2</sub> = 2

value<sub>e11</sub>

value<sub>e12</sub>

value<sub>e21</sub>

value<sub>e22</sub>

parameter

value

x<sub>1</sub> = 1

x<sub>1</sub> = 2

x<sub>1</sub> = 3

x<sub>1</sub> = 4

x<sub>1</sub> = 5

value<sub>e1</sub>

value<sub>e2</sub>

value<sub>e3</sub>

value<sub>e4</sub>

value<sub>e5</sub>

(c) 1-dimensional

x<sub>1</sub>

...

x<sub>n-1</sub>

x<sub>n</sub> = 1

x<sub>n</sub> = 2

value<sub>e1...11</sub>

value<sub>e1...21</sub>

value<sub>e1...11</sub>

value<sub>e1...21</sub>

value<sub>e1...11</sub>

value<sub>e1...12</sub>

value<sub>e1...22</sub>

value<sub>e1...12</sub>

value<sub>e1...22</sub>

value<sub>e1...12</sub>

(d) n-dimensional (flattened)

such a location, trying to read it will not result in an exception but will return an `undef` value (a special yet valid value representing that the value at the requested location is not defined). In Table 2.1 we can see the discussed illustration of basic functions. The corresponding tables have the same number of dimensions as there are parameters of the concrete function. We can always flatten the illustration in case of more than two dimensions as shown in Table 2.1d. Those tables are possibly infinitely big, while even the number of parameters is not fixed — every possible combination of parameter values (locations) is valid. This has to be kept in mind when there is the intention to simulate such a high-level design and one has to be aware of related drawbacks.

**Derived functions** are meant for computing a value based on a fixed scheme [2], implementing a part of the business logic in the designed system. Therefore those functions are not updatable. They may use other basic functions to compute their value, but they may not update any of them.

**Rules** are by definition of form [2]:

```
1  if [condition] then [updates]
```

We can also encapsulate a set of such transition rules into a named rule. An update is then updating a concrete location with a value from another location or a computed value using derived functions.

## 2.1 Notation and conventions

The ASM notation is very flexible. In this section we show the notation we have chosen for this thesis with the intention to improve readability for a wide spectrum of readers, from technical people, e.g., developers, software engineers, system architects, system designers, to business people.

### 2.1.1 Universes and types

Although ASMs are not typed by default, all values belong to one or more universes. Universes are based on set theory [79], which fits the needs of high-level specifications especially on business level of abstraction, where hand proofs are more usual and a type system may introduce unnecessary complexity and reduce flexibility of refinements [80]. Next to meta-model related universes there are also some standard ones, e.g., NUMBERS or STRINGS. Universes do not facilitate any particular structuring and every universe contains a special value `undef` for Locations, which have not yet been defined as discussed earlier in this chapter, reducing the initialization overhead. When working with a collection of elements the default structure used here is a set. Nevertheless, also multiset and their order counterparts, ordered set and list may be used. To distinguish between those four concepts we adopt the notation from [81] where:

**set** will be displayed as contained by curly bracket separated by spaces (see Equation 2.1). If there is any confusion arising from spaces being insufficiently clear, we will use the underscore to represent a space. In ASM code the modifier `SET` will be used to denote that the input or output parameter will be set.

$$s = \{abcd\} = \{bcad\} = \{b\_c\_a\_d\} \quad (2.1)$$

**ordered set** will be written, as a set, between curly brackets but separated by commas (see Equation 2.2). In ASM code the modifier `ORDEREDSET` will be used to denote an ordered set.

$$os = \{a, b, c, d\} \quad (2.2)$$

**multiset** will be written between square brackets separated like sets by spaces or underscore as for sets (see Equation 2.3). In ASM code the modifier `MULTISET` will be used to denote a multi set.

$$ms = [abbc d] = [bcba d] = [b\_c\_b\_a\_d] \quad (2.3)$$

**list** will be displayed as contained by square brackets with the elements separated by commas (see Equation 2.4). In ASM code the modifier `LIST` will be used to denote a list.

$$l = [a, b, b, c, d] \quad (2.4)$$

All modifiers can optionally take one parameter identifying the universe the elements in such structure will be from. E.g., `SET(SEQUENCEFLOWS)` denotes a set with its elements being from the universe `SEQUENCEFLOWS`. All the modifiers are reserved key words and therefore no universe can take a name `SET`, `ORDEREDSET`, `MULTISET` or `LIST`.

Types on the other hand may be relevant on the technical level of abstraction of the ground model. Important reasons to use a type system, especially on lower levels of abstraction, are among others to facilitate structuring of data of the designed system and to have a type checker with the ability to automatically detect errors at

compile time [82]. While those reasons may become crucial on the technical level they are usually not relevant on the business level of the design, where the set theory provides a powerful and simple alternative [80].

Universes are used here to define simple relations between structures [39] on higher levels and types as defined in [82] may be further used on lower levels. Universes and types are written in uppercase letters and the underscore symbol “\_” is used to separate words within the identifier. We assume, if not explicitly stated otherwise, that a camel case singular version of a universe is a member of such a universe, as shown in Equation 2.5.

$$connectingObject \in CONNECTING\_OBJECTS \quad (2.5)$$

### 2.1.2 Basic functions

In ASMs functions are divided into two main groups: basic functions and derived functions. The difference between those two types is enormous. The first can be seen as a table (see Table 2.1), where the name of the function is the name of the table and its parameter combination is a concrete location with a value. Any parameter combination, i.e., every location will yield a value. In case no update was ever made to the particular location, the yielded value will be by default `undef`, i.e. no exception such as “undefined value” will occur. This is default ASM behavior [2] defined using `reserve` in [39]. There are the following access types of basic functions, according to [2]:

**static** are used as constants. Their values do not contribute to the state and they cannot be changed by any rule of the designed system. We use the reserved keyword `static` in their signatures to identify static functions.

**monitored** functions contain read-only locations and are identified using the reserved keyword `monitored` in their signatures. Their values can be accessed by the rules and derived functions of the designed system, but cannot be updated (cannot appear as leftmost functions) in the rules of the designed system. They may be updated by the environment or by another machine.

**out** are the opposite of monitored functions. The rules of the designed system may update the locations of such functions but they may not read their content - they can appear only as leftmost functions in updates of the designed system. They are identified using the reserved keyword `out` in their signatures.

**controlled** are identified by the reserved keyword `controlled` and can be read and updated by the rules of the designed system but cannot be updated by the environment or by another agent.

**shared** can be read and updated by both, the designed system and the environment. They are meant for interaction between the designed system and the outside world. Those functions are identified by the reserved keyword `shared` in their signatures.

The signature of a basic function (see listing 2.1) is composed of the reserved keyword representing the type of the function, the name of the function and its input/output parameter universes. We use camel case encoding of the name of the function in case the name is composed of multiple words, where the function name always begins with a lowercase letter.

---

**Listing 2.1** Basic function signature

---

```
1 monitored someFunctionName :  $U_1 \times \dots \times U_n \rightarrow R_1 \times \dots \times R_m$ 
```

---

### 2.1.3 Derived functions

Derived functions are the second discussed main type of functions. Instead of a table lookup the value of a derived function is computed using other basic or derived functions based on a fixed scheme we call the body. If such a body is not defined, we say that such a derived function is abstract. Derived functions cannot be updated and they cannot update any location. Updates may be caused only by rules (see section 2.1.4). The signature of a derived function is based on same rules as

---

**Listing 2.2** Derived function signature

---

```
1 derived someDerivedFunctionName :  $U_1 \times \dots \times U_n \rightarrow R_1 \times \dots \times R_m$ 
```

---

the signature of basic functions, using the reserved keyword `derived` to identify derived function. The returning value is then returned using the `return` statement (see listing 2.3). We also allow implicit guards further explained in section 2.1.4.

---

**Listing 2.3** Derived function definition

---

```
1 derived someDerivedFunctionName :  $U_1, \dots, U_n \rightarrow U_R$ 
2 derived someDerivedFunctionName( $p_1, \dots, p_n$ ) =
3   // computation of the value ...
4   return result
```

---

### 2.1.4 Rules

A rule or also transition rule has the general form shown in listing 2.4, where  $c$  is

---

**Listing 2.4** General form of a rule

---

```
1 if  $c$  then  $u_1$  else  $u_2$ 
```

---

the condition or also guard of the update  $u_1$ , and alternatively update  $u_2$  in case  $c$

---

**Listing 2.5** Rule signature

---

```
1 rule SomeRuleName :  $U_1, \dots, U_n$   
2 rule SomeRuleName( $p_1, \dots, p_n$ ) =  
3 // set of updates
```

---

does not hold [39, 2]. In general, any location update is a rule [83]. A rule in general represents a set of guarded updates as shown in listing 2.4. The signature of a rule is identified by the reserved keyword **rule** and may be parametrized (see listing 2.5). We use camel case encoding of the name of the rules in case the name is composed of multiple words, where the rule name always starts with an uppercase letter. The parameters in the signature are following the same rules as for derived functions. If universes are used with the parameters, they will create an implicit guard, which would otherwise have to be explicitly defined as shown in listing 2.6.

---

**Listing 2.6** Implicit guard in a rule signatures

---

```
1 rule SomeRuleName :  $U_1, \dots, U_n$   
2 rule SomeRuleName( $p_1, \dots, p_n$ ) =  
3 if  $p_1 \in U_1 \wedge \dots \wedge p_n \in U_n$  then  
4 // set of updates
```

---

### 2.1.5 Auxiliary constructs, statements and keywords

In this section we introduce used auxiliary constructs, which are not part of the original ASM notation [2], such as **choose**, **forall**, but often occur in the literature. For the complete list of all constructs, the default ones from the original ASM notation and the auxiliary ones, see the glossary on page 153.

**return** defines a keyword causing execution to leave the current rule or derived function. It can also be used as a construct in the form of **return result in body**, where the initialized variable *result* will contain the returned value after the *body* block has been executed. In some literature this construct is used to prevent jumping in the pseudo code improving sequential reading of the code.

**add element to set** is a construct, which adds an *element* to a *set*.

**remove element from set** is a construct, which removes an *element* from a *set*.

**seqblock** construct indicates a block, where all statements are executed sequentially, as if there would be a **seq** statement between each of them [2, sec. 2.4.3].

**parblock** construct indicates a block, where all statements are executed in parallel, as if there would be a **par** statement between each of them [2, sec. 2.4.3]. This is the default behavior of ASMs in case no sequential or parallel execution is explicitly defined.

foreach *element* with  $\varphi$  do *R* will iterate and fire rule *R* for each *element* satisfying  $\varphi$ . This is formally an abbreviation to the following:

```
1 local elements  $\leftarrow \{ e \mid \varphi(e) \}$  in  
2   while |elements| > 0 do  
3     choose element  $\in$  elements do  
4       remove element from elements  
5       R
```



*Choose mnemonic identifiers. If you can't remember  
what mnemonic means, you've got a problem.*

— Larry Wall

## Chapter 3

# The refinement method

The ASM Method allows describing a system formally on a certain abstraction level [2, 3, 4, 5]. It supports the system development process starting with the problem specification through several steps till the implementation by incrementally refining the system model. This basic approach is frequently used, e.g., the Information Technology Infrastructure Library (ITIL) [84] life-cycle or typical software development process models have much in common. System development processes start with an initial problem specification typically followed by analysis, requirements capture, design and implementation steps. Depending on the process model, these steps occur in a linear sequence, e.g., the linear sequential model, or in an evolutionary, iterative manner [85].

Starting with an abstract model of the system, it is continuously refined within the ASM method, i.e., enriched and thus made more specific. The typical tool for developing models with the ASM method is a text editor. All models are described in documents, consisting of formal descriptions and natural language explanations. Typical document types are a project proposal, a system specification and various design documents. They all refer to the same system under design but have different purpose, and thus differ in the abstraction level and the target audience. When the system needs to be changed, it is very hard to keep the models consistent over the abstraction levels, described in several documents using specific conventions designed for different target audiences. Thus, some kind of relation or linking between the corresponding modeling elements on the different abstraction levels is needed.

We illustrate the problem by an example in software development, which is a typical domain to deploy the ASM method. Two main levels of abstraction are defined in this context, the business level and the technical level of abstraction. The first focuses on the needs of business people while technical people are the target audience for the latter. We do not deal with the differences between formal and informal methods here. We assume that all models on all abstraction levels are specified using the ASM method. The ideas presented here are not limited to software development and can be also adapted to other domains.

Section 3.1 shows how to systematically specify a system on an abstract level (namely the business level), to provide a readable natural language description for

the target audience and be prepared for consistent vertical refinements as well. Most of the model refinement steps are part of the technical abstraction level, which is described in section 3.2 with special emphasis on the structure and description elements used for specifying basic functions and rules. The developed 4-step transition approach is described in section 3.3. Section 3.4 shows how to assure consistency in the case of changes in one of the abstraction levels.

### 3.1 Business level of abstraction

The specification of a system starts on the business level. Let's assume that the intended documents of a system are: project proposal, project specification and technical specification. Then the first two, the project proposal and the project specification, fit the business level of abstraction described here. The assumed audience for this type of documents is *technically aware business people*.

The development process of a system usually starts with a project proposal provided by the *product owner*. This project proposal is a starting point for the *software analyst* who will capture the requirements and write a document representing, as said in [15]:

“[...] a succinct process-oriented model of the to-be-implemented piece of *real world*, transparent for both the customer and the software designer so that they can serve as the basis for the software contract, a document which binds the two parties involved.”

The step between the project proposal and the analysis document may be considered as one of many refinement steps. Those are the main common documents written on the first level of abstraction, the business level.

#### 3.1.1 Business level document structure

A formal specification on business level includes a significant amount of abstract rules and abstract derived functions. The amount of those abstract functions and rules will decrease during the refinement steps, even on this level of abstraction. However the goal on this level is not a complete elimination of those abstract rules or abstract derived functions. Also a significant amount of other concrete details, e.g., data types, may be kept abstract on this level of abstraction. The set theory will provide a simple and powerful alternative, avoiding restrictions coming with a type system [80]. The resulting documents usually tend to be natural language documents rather than a collection of pseudo code. The formal core of the description should be captured in the ASM description, still additional informal description frequently occurs. An example of such a description can be seen in [15, 16], where the BPMN 2.0 Specification [1] is described formally. We will use and build on those examples here.

### 3.1.2 Using formal elements in an informal text

The document may look like rather a continuous text with formal elements embedded. Blocks of pseudo code are present but rather small, using a significant number of abstract derived functions and abstract rules. Consider the example in [16] and assume we need to get the information, which process instance a concrete token [86] is in. For this purpose we may introduce an `instanceOfToken(t)` function which will provide us with the required information. We may want to get the same information for running processes or instantiated activities. For this purpose we introduce, in the same way, two more functions: `instanceOfProcess(p)` and `instanceOfActivity(a)`. In those proposed conventions all three parameters, `t`, `p` and `a` are arbitrary structures (sometimes also called Tarski structures) with no defined type yet. This way we can speak about instances of tokens, processes or activities in an informal text with minimal impact on the fluency and naturalness of the text.

In order to introduce such a formal element without a type, those elements have to be explained. One possibility is to introduce some conventions or guidelines. In our example we introduced a “*propertyOfSubject*” naming convention for basic and derived functions to preserve the naturalness of the text. A similar “*ActionOfSubject*” naming convention can be introduced for rules. This will lead to renaming of rules shown in [15, 16] to Equation 3.1 and Equation 3.2.

$$\text{WorkflowTransition} \Rightarrow \text{TransitionOfWorkflow} \quad (3.1)$$

$$\text{GatewayTransition} \Rightarrow \text{TransitionOfGateway} \quad (3.2)$$

The intention is to introduce semantics into the naming conventions. This enables us to use formal elements in natural texts while maintaining the option to keep most of the properties abstract, including types. This is an important aspect especially at the beginning of the modeling process. Those conventions enable the transition described in section 3.3 and also help to preserve the consistency between the abstraction levels described in section 3.4. The concrete conventions are flexible and up to the designer of the system in question. The examples here are only one possible option.

Similarly we would write a function or rule with multiple parameters. E.g., a function name, which represents all tokens in a sequence flow and a certain process instance, using the introduced conventions, in Equation 3.3.

$$\text{tokensInSequenceFlowAndInstance}(a, i) \quad (3.3)$$

### 3.1.3 Defining universes

One of the refinement steps is to start defining universes of the data structures. This may be one of the differences between a project proposal and the project specification. Consider the functions mentioned in the previous section 3.1.2. Now it may make sense to state that each of them is meant for a different data type by refining their signatures with universes, e.g., `instanceOfToken : TOKENS`

→ INSTANCES for getting the instance of a token, `instanceOfProcess` : - PROCESSES → INSTANCES for getting the instance of a process and `instanceOfActivity` : ACTIVITIES → INSTANCES for getting the instance of an activity. Using the ASM method two signature types are available:

**Declaration signature** used for declaring rules or functions, their parameter universes and return value universes.

**Definition signature** used for defining, calling or referring to a rule or function. Rules and derived functions may also define their bodies after this type of signature.

Where the parameter universes are not yet defined, they may be discovered using the naming conventions mentioned in section 2.1 and section 3.1.2. This allows using formal elements in natural text as described in section 3.1.2 without breaking the fluency or naturalness of the text.

The model can be refined by adding more formal information to the description. The implicit semantics within the names of the mentioned functions are now explicitly expressed. The properties of these universes are still abstract and so are relations between them. After several refinement steps some of these relations or properties may appear obvious, but their explicit formal definition is intended to be done in the technical level. In the implementation these universes will be the basis for types, which may be introduced even in the system specification on some of the lowest levels of the technical abstraction level.

## 3.2 Technical level of abstraction

Generally, after the product owner and the software designer agree on the model, the software designer proceeds with a *technical specification* of the intended system. Such a specification can pass through several refinement steps before the first implementation attempt. Also after the first release is deployed additional refinement steps may occur as change requests [87]. The direction is to already make more product specific decisions and go into more detail. Yet still concrete implementation decisions do not have to be made. The model can still be kept abstract enough to keep the implementation language and similar questions open. But the emphasis should be already put on technical, *abstract-as-possible* and *complete-as-necessary*, aspects.

The specification already went through several refinement iterations before being transformed to this level of abstraction. The project proposal or similar document from the business level is the basis for the technical specification. But there is no formal link between those documents on the different abstraction levels, which would preserve the consistency of both of them. The resulting two documents are typically kept concurrent till the first implementation release, accepted by the *product owner*, is deployed. Later the two documents detach from each other. The technical specification may live further and be refined as a consequence of Request for Change (RFC) while the product specification becomes obsolete.

### 3.2.1 Technical level document structure

The emphasis passes from the natural text document described in section 3.1 to rather pseudo code constructs. The complexity of the pseudo code constructs increases and the amount of abstract functions and rules decreases. Yet, still not all of them have to be eliminated. Universes or even types in this abstraction level may be mostly defined at some point, so may be their properties and relations. The usage of universes or types may lead to the refinement of names of some functions or rules. Assume the functions mentioned in section 3.1.3. The consequence of the change of abstraction level from business level to technical level may lead to cutting of parts of the function or rule names. Based on the proposed naming convention refinement, those name parts are superfluous since the *property* and *subject* are already formally defined using types, e.g.:

- `instanceOfToken` : `TOKENS`  $\rightarrow$  `INSTANCES`
- `instanceOfProcess` : `PROCESSES`  $\rightarrow$  `INSTANCES`
- `instanceOfActivity` : `ACTIVITIES`  $\rightarrow$  `INSTANCES`

may lead to:

- `instanceOf` : `TOKENS`  $\rightarrow$  `INSTANCES`
- `instanceOf` : `PROCESSES`  $\rightarrow$  `INSTANCES`
- `instanceOf` : `ACTIVITIES`  $\rightarrow$  `INSTANCES`

With this refinement, function and rule overloading is enabled. To still use these functions in natural language text, we have to be more concerned about the parameter naming conventions than we were in the business level. E.g., `instanceOf(t)` may lead to confusion about the parameter type. Thus the usage of `instanceOf(token)` in natural language text is recommended. Also the relations between the different universes or types can be defined, which introduces a structure into types [15], e.g., as shown in Equation 3.4.

$$\text{NODES} = \text{ACTIVITIES} \cup \text{EVENTS} \cup \text{GATEWAYS} \quad (3.4)$$

### 3.2.2 Selectors and properties of universes

An analogy between function application and selection, described in [88] and shown in Equation 3.5, may be used to introduce namespaces and a way how to define properties of universes.

$$f(x) \equiv x.f \quad (3.5)$$

Using this analogy we may refine the functions such as `instanceOf` : `TOKENS`  $\rightarrow$  `INSTANCES`, `instanceOf` : `PROCESSES`  $\rightarrow$  `INSTANCES` or `instanceOf` : `ACTIVITIES`  $\rightarrow$  `INSTANCES` and write them as shown in listing 3.1.

Obviously this is a less natural language text oriented approach than the versions in section 3.1. However, the link between all the versions can be preserved. Furthermore, we now can use universes as namespaces to functions as shown in listing 3.2.

---

**Listing 3.1** Selector notation

---

```
1 TOKENS.instance → INSTANCES
2 PROCESSES.instance → INSTANCES
3 ACTIVITIES.instance → INSTANCES
```

---

---

**Listing 3.2** Namespaces

---

```
1 TOKENS {
2     instance → INSTANCES
3     sequenceFlow → SEQUENCE_FLOWS
4 }
```

---

This is just another way of how to write the previous definition of a function, but also introduces a way to define properties of universes and to aggregate functions into logical units. We are simply defining a property `instance` and `sequenceFlow` for any data structure from the universe `TOKENS`. Additionally the name of the functions is refined by cutting of the “Of” reserved separator from the function names, as it lost its semantical meaning when the parameter was transformed to a selector. Those refinements allow creating more technical oriented descriptions, decreasing the ambiguity, since the informal parts will be reduced by transforming them into the bodies of derived functions and rules.

In case of a function with multiple parameters, the first parameter may be used as a selector and the rest as parameters of the function as shown in Equation 3.6.

$$f(x, y, z) \equiv x.f(y, z) \quad (3.6)$$

The function `tokensInSequenceFlowAndInstance(a, i)` introduced in section 3.1.2 will be reformatted to Equation 3.7.

$$\text{SEQUENCE\_FLOWS.tokens} : \text{INSTANCES} \rightarrow \text{SET}(\text{TOKENS}) \quad (3.7)$$

Also note that at this level of abstraction both notations, with or without selectors, are considered equal. Both of them can be used at the same time in the technical level document.

### 3.3 Transition between the abstraction levels

The transition between the abstraction levels, from business level to technical level, can be (semi-)automatic. The idea of such a transition is to introduce transition rules, which will enable extracting the formal parts of the document on the technical level and generate the skeleton for the technical level document. A manual interaction may be necessary depending on the abstraction degree of the business level. The ability of (semi-)automatic transition heavily depends on the conventions used on the business level discussed in section 3.1 and section 3.2.

The transition between abstraction levels can be regarded as a specific type of refinement, where one function is replaced by another. The transition between the abstraction levels here can be seen as a way of a syntax refinement. Equation 3.8 and Equation 3.9 for example show the refinement of adding universes to the specification.

$$\begin{aligned} \text{instanceOfToken}(t) &\Longrightarrow_{T_1} \\ &\Longrightarrow_{T_1} \text{instanceOfToken} : \text{TOKENS} \rightarrow \text{INSTANCES} \end{aligned} \quad (3.8)$$

$$\begin{aligned} \text{arcOfToken}(t) &\Longrightarrow_{T_1} \\ &\Longrightarrow_{T_1} \text{arcOfToken} : \text{TOKENS} \rightarrow \text{SEQUENCE\_FLOWS} \end{aligned} \quad (3.9)$$

The naming refinement is then shown in Equation 3.10 and Equation 3.11.

$$\begin{aligned} \text{instanceOfToken} : \text{TOKENS} \rightarrow \text{INSTANCES} &\Longrightarrow_{T_2} \\ &\Longrightarrow_{T_2} \text{instanceOf} : \text{TOKENS} \rightarrow \text{INSTANCES} \end{aligned} \quad (3.10)$$

$$\begin{aligned} \text{arcOfToken} : \text{TOKENS} \rightarrow \text{SEQUENCE\_FLOWS} &\Longrightarrow_{T_2} \\ &\Longrightarrow_{T_2} \text{arcOf} : \text{TOKENS} \rightarrow \text{SEQUENCE\_FLOWS} \end{aligned} \quad (3.11)$$

Equation 3.12 and 3.13 show the transition between function application and selection, if types of parameters are already defined. Basically we collect all functions and group them by the type of the first parameter and then do the transition.

$$\begin{aligned} \text{instanceOf} : \text{TOKENS} \rightarrow \text{INSTANCES} &\Longrightarrow_{T_3} \\ &\Longrightarrow_{T_3} \text{TOKENS.instance} \rightarrow \text{INSTANCES} \end{aligned} \quad (3.12)$$

$$\begin{aligned} \text{arcOf} : \text{TOKENS} \rightarrow \text{SEQUENCE\_FLOWS} &\Longrightarrow_{T_3} \\ &\Longrightarrow_{T_3} \text{TOKENS.arc} \rightarrow \text{SEQUENCE\_FLOWS} \end{aligned} \quad (3.13)$$

Grouping such function signatures ( $T_4$ ) will then result in listing 3.2. This example illustrates the 4-step transition approach ( $T_{1,2,3,4}$ ) proposed here. Neither the number of steps nor their definition should be considered complete. Both, the number of steps and their definition may be bent to the specific needs. The placement of a concrete refinement step in an abstraction level class, business level or technical level, also should not be considered fixed. Here we expect the 1<sup>st</sup> step ( $T_1$ ) to be done on the business level and the 2<sup>nd</sup>, 3<sup>rd</sup> and 4<sup>th</sup> ( $T_{2,3,4}$ ) ones on the technical level.

The intended (semi-)automatic transition between the two levels of abstraction will apply those transition rules to existing formal objects in the business level documentation. Based on their names, the types of parameters and return values can be derived. We assume that the function and rule names in the business level document are based on the naming conventions discussed in section 3.1.2. We demonstrate a basic set of such transition rules in terms of syntax analysis [89] and the result can be seen in listing 3.3

---

**Listing 3.3** Definition of lexical structure for formal objects in a document

---

```
1 # Reserved words for separator terminals
2 s ← ("A" | "An" | "And" | "For" | "From" | "Given" | "In" | "Of" | "On"
3     | "Through" | "To" | "Using")
4 l ← /[a-z][a-zA-Z]*/ # Terminals starting with a lower case letter
5 u ← /[A-Z][a-zA-Z]*/ # Terminals starting with an upper case letter
6
7 S → F | R # Starting rule of the production rules
8 F → l (s U) # Function non-terminal
9 R → u (s U) # Rule non-terminal
10 U → u # Universe non-terminal
```

---

### 3.4 Preserving consistency between the abstraction levels

Preserving consistency will pay itself off as soon as the product owner will apply an RFC to the original proposal. Usually such an RFC [87] is submitted as a distinct document. Using the conventions discussed in this thesis the product owner will be able to make changes to the original proposal and the technical level specification skeleton may be (semi-)automatically updated.

The idea of preserving the consistency between the business level and technical level relies on the transition rules discussed in section 3.3. After an update in the business level, the technical level document skeleton has to be updated using the mentioned transition rules. New formal objects from the business level will be added to those in the technical level (semi-)automatically. Furthermore, formal objects which are no longer available in the business level are marked, not deleted, in the technical level for further handling. The product designer will see all the differences and changes, made by the product owner, in the technical level specification and will refine the changes towards the implementation. A consistency test is also possible vice-versa, starting at the technical level up to the business level.



*A computer lets you make more mistakes faster than any other invention, with the possible exceptions of handguns and Tequilla.*

— Mitch Ratcliffe

# Part II

## **Business Process Model and Notation**

In this part the Enhanced Business Process Model and Notation (BPMN<sub>e</sub>) will be introduced. The advantages and improvements of BPMN<sub>e</sub> compared to Business Process Model and Notation (BPMN) will be discussed. Furthermore, the BPMN specification will be formally described using the Abstract State Machine (ASM) method.



*The nice thing about standards is that there are so many of them. If you don't like one, just wait for next years model.*

— Andrew Tanenbaum

## Chapter 4

# Modeling elements in BPMN

The Business Process Model and Notation (BPMN) with current version 2.0 [1] is well established for describing business processes. With September 2013 BPMN has become an ISO/IEC-Standard (reference number 19510:2013) [6]. Still the standard has some remaining ambiguities and inconsistencies. While participating in a project with the focus to formally describe the BPMN standard we dealt with those inconsistency problems and improved the meta-model structure to enable better reusability. Those problems appear in both, the vertical refinements on one hand and on horizontal extensions to BPMN on the other.

The motivation for this chapter is to evaluate the Business Process Model and Notation (BPMN) meta-model and identify problems described above, propose improvements and show the influence of our revisions on the system design. The goal is to make the meta-model more compact and graspable. The intended modifications will be in favor of future refinements in the vertical direction and extensibility of the meta-model in the horizontal direction. With vertical refinements we address the process starting with the high-level meta-model, which will continuously be refined by adding more concrete specifics incrementally resulting in an implementation using the stepwise refinement method described in chapter 3. Such refinements should happen without any gap between the original high-level specification and the implementation. By horizontal extensibility we contemplate the ability of a model to be easily and consistently extended. In the specific case of the BPMN the focus is to enable the introduction of additional attributes to existing elements or even enrich the modeling language with new elements, while preserving the consistency of the modeling language and the underlying meta-model.

Further, we will elaborate through the BPMN 2.0 meta-model [1] and separate the existing graphical elements into two groups: *basic elements* and *compound elements*. The target is to have the *basic* set of graphical elements as small as possible and to be able to compose all *compound elements* from *basic elements*. This will be used as a base for the decomposition further in this thesis. The behavior building block will be extracted from the basic elements as mentioned in section 1.5. Such behavioral building blocks will be further used in chapter 6 to compose the formal description of the standard.

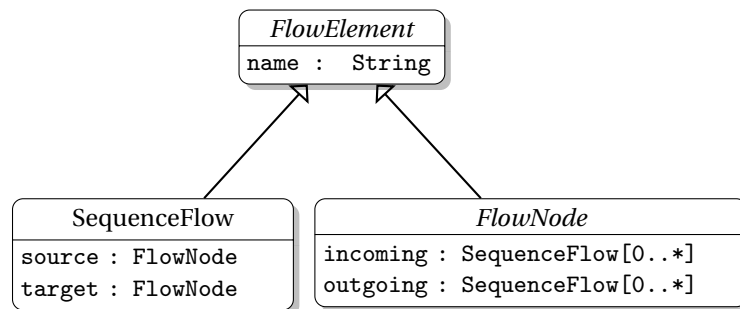
Proposals in this chapter should not affect the modeling scope of the BPMN 2.0 specification. The enhanced meta-model introduced further in chapter 5 and formalized in chapter 6 should accept any workflow model based on the BPMN meta-model. Backward conversion should be also possible, but with loss of introduced enhancements. E.g., the introduced sole exclusive gateway in section 6.6 would be converted back to just an exclusive gateway and lose its additional semantics. We will only concentrate on the parts of BPMN 2.0 specification which are specifically used in this thesis. Refinements of the rest of the BPMN 2.0 specification is left for some future work, see section 9.2.

This chapter is structured as follows. In section 4.1 the basics of flow elements will be discussed, followed by a section for each flow node type: section 4.2 for activities, 4.3 for events, and section 4.4 for gateways. Each of those sections will then be divided into subsections discussing the flow node type, the control flow regarding the flow node type, and specifics of the flow node type.

## 4.1 Flow elements

There are several sub classes of the *FlowElement* meta-model class in the BPMN 2.0 specification [1], but we will limit those to the relevant ones for this work, which are: the *SequenceFlow* class and the *FlowNode* class as shown in Figure 4.1. We omit the `isImmediate` attribute since this is not applicable for executable processes, on which concentrate here. We also omit the `conditionExpressionOfSequenceFlow` attribute for this class, as we will model this using the gateway behavior building block as mentioned above. The remaining attributes of the *SequenceFlow* and *FlowNode* class are just for referencing the instances of each others class, i.e., the incoming and outgoing sequence flows of a flow node or the target and source flow node of a sequence flow. For now we leave them the way they are. Later we will model some as basic functions, which will actually store the reference, and the others as derived functions to preserve consistency in the locations of our ground model.

**Figure 4.1** *FlowElement* class and its immediate relevant children *SequenceFlow* and *FlowNode*.



In the case a flow node is a source or target of multiple sequence flows, there are two basic conceptual options how to treat such flows:

- as alternative paths, or
- as parallel paths

In the case of alternative paths, the condition to fire the target flow node is that a token is present on one (XOR - exclusive gateway), or one or more (OR - inclusive gateway) of the alternative paths. In contrast to that, if a flow node is a target of multiple parallel paths, a token has to be present on all (AND - parallel gateway) such incoming sequence flows in order to fire the target flow node. We can see that there is an overlapping space where we cannot decide if the incoming or outgoing paths are alternative or parallel, without knowing the additional context, e.g., the type of the flow node.

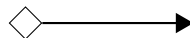
Also in case of alternative paths, a token will be put on one (XOR - exclusive gateway), or one or more (OR - inclusive gateway) outgoing sequence flows. In case of parallel paths, a token will be put on all (AND - parallel gateway) outgoing sequence flows. This will be further used for the refined decomposition of BPMN<sub>ε</sub> in chapter 5.

In section 4.2 the number of incoming and outgoing sequence flows for activities will be cut down to one in each direction. This has the consequence that the conditional sequence flow notation shown in Figure 4.2 is no longer needed, since the splitting to alternative paths is then realized explicitly by a gateway behavior (see Figure 1.3) - every conditional flow may be *compounded* using a gateway. This correlates with the behavior decomposition idea mentioned in section 1.5, where a pure gateway behavior incorporates exactly this decision making using conditions. Therefore a conditional flow will be decomposed as if it is an inclusive gateway with one outgoing sequence flow.

---

**Figure 4.2** Conditional flow

---




---

As described in [1, p. 34]:

Uncontrolled sequence flow refers to flow that is not affected by any conditions or does not pass through a gateway. The simplest example of this is a single sequence flow connecting two activities. This can also apply to multiple sequence flows that converge to or diverge from an activity. For each uncontrolled flow a token will flow from the source flow node through the sequence flows to the target flow node.

As *simple* activities (see section 1.5) can have only one incoming and one outgoing sequence flow this uncontrolled flow applies only for the simplest example where

it is connecting two activities. Every real flow node incorporates a gateway behavior and therefore uses controlled sequence flow. With this approach we intend to solve issues, i.e., `ProducingTokenOnSequenceFlowForInstance` for an exclusive gateway where conditions of more outgoing sequence flows may hold.

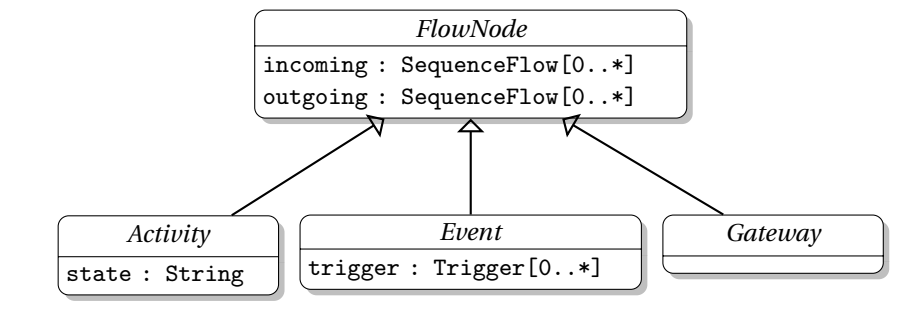
## 4.2 Activities

The following section will examine the activity flow node from the BPMN specification.

### 4.2.1 Activity flow node

The *Activity* flow node class is a direct descendant of the *FlowNode* class in the meta-model as shown in Figure 4.3. This meta-model class contains the `lifeCycleStateOfActivityInInstance` of each instance [1, tab. 10.4, fig. 13.2]. We omit the following attributes from the original meta-model: the optional `defaultSequenceFlowOfActivity` attribute identifying the default sequence flow that should receive a token in case the activity is a source of multiple outgoing conditional flows and none of the `conditionExpressionOfSequenceFlows` of those conditional flows hold. This is a consequence of the refinement of the `conditionExpression` attribute of the *SequenceFlow* meta-model class in section 4.1 and using the `gateConditionForSequenceFlow` instead. We will show in chapter 6 how those behaviors regarding the `defaultSequenceFlowOfActivity` and the `gateConditionForSequenceFlow` are modeled to preserve the intended functionality of BPMN. With this we also eliminated duplicates since this attribute is also defined for exclusive gateway, inclusive gateway and complex gateway in the BPMN 2.0 [68, tab. 10.123, 10.124 and 10.125].

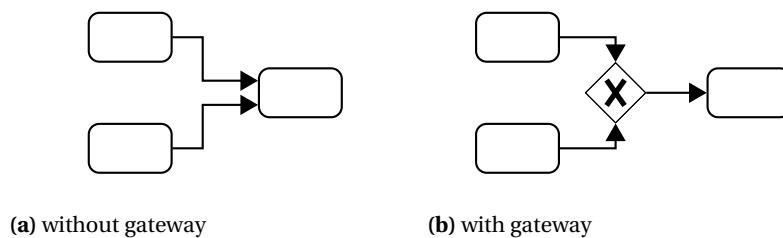
**Figure 4.3** *FlowNode* class and its immediate relevant children *Activity*, *Event*, and *Gateway*



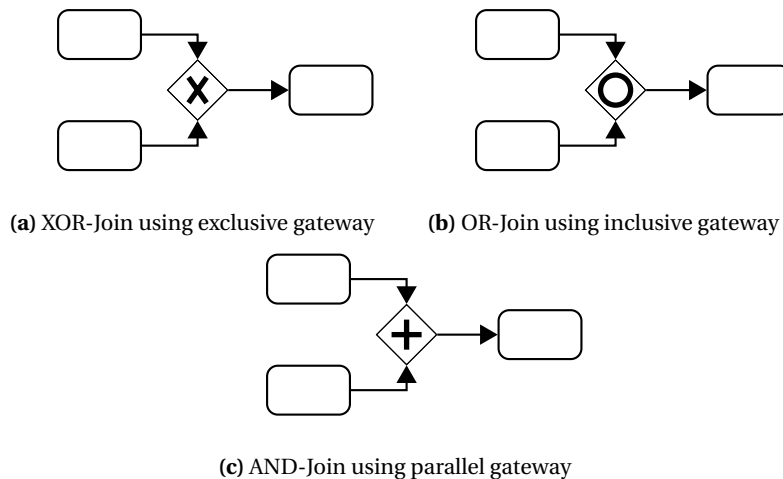
### 4.2.2 Activity control flow

According to [1, p. 153] an activity may be a target for multiple incoming and source for multiple outgoing sequence flows. In the case of multiple incoming sequence flows they may be from alternative or parallel paths. If a token arrives on one of the paths, the activity will be instantiated immediately. This is considered a so called uncontrolled flow and can be also accomplished (simulated) using a controlled flow with a gateway. *Merging Extended Modeling Elements* (see Figure 4.4) cp. [1, p. 38]

**Figure 4.4** Merging extended modeling element



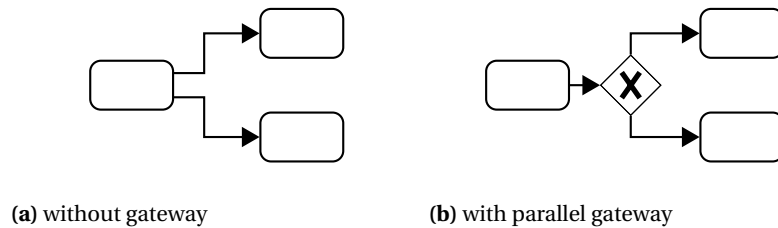
**Figure 4.5** Merging using gateways explicitly



describe the case where all incoming sequence flows to an activity are coming from alternative paths and a gateway is not needed. This is excluding the option of an activity being a target of multiple parallel sequence flows mentioned earlier. Also as mentioned at the beginning of this chapter we are identifying *Basic Elements*, which *Extended Modeling Elements* are not part of. The decision of how many of those

paths will be used is implemented inside the Activity meta-model class. Since this is already implemented in gateways, by omitting merging without a gateway, the implementation can be simplified and this duplication can be removed.

**Figure 4.6** Fork extended modeling element



To make this problem clear and easier to implement, we follow the best practice patterns [33] and restrict a *simple* activity to have exactly one incoming sequence flow. If multiple incoming alternative or parallel paths are targeting an activity, this should be modeled using gateways. The gateway type will make clear if the incoming sequence flows are built by alternative or parallel paths. This will replace the merging element described in [1, p. 38] with different join elements (*AND-Join*, *OR-Join*, *XOR-Join*). The case where an activity is the target of multiple incoming parallel paths, also known as *AND-Join*, is shown in Figure 4.5c. The case where an activity is the target of multiple alternative paths can be split into two cases:

**Exclusive alternative paths** also known as *XOR-Join*, is the case, where a token has to be present on exactly one alternative path, in order to fire the target activity (see Figure 4.5a).

**Inclusive alternative paths** also known as *OR-Join*, is the case, where a token has to be present on one or more alternative paths in order to fire the target activity (see Figure 4.5b). This way also the uncontrolled flow can be simulated.

According to [1, p. 153] an activity may be a source of multiple outgoing sequence flows and if so, these paths are treated as parallel paths (see Figure 4.6 described in [1, p. 36]). To make it consistent with the previous simplification about just one incoming sequence flow, all activities can be a source of only one outgoing sequence flow. If more alternative or parallel outgoing paths are needed that should be modeled using gateways as well. The default case, where the activity is a source for multiple parallel paths is shown in Figure 4.7c. The other options are *Exclusive Extended Modeling Element* shown in Figure 4.7a and *Inclusive Extended Modeling Element* shown in Figure 4.7b.

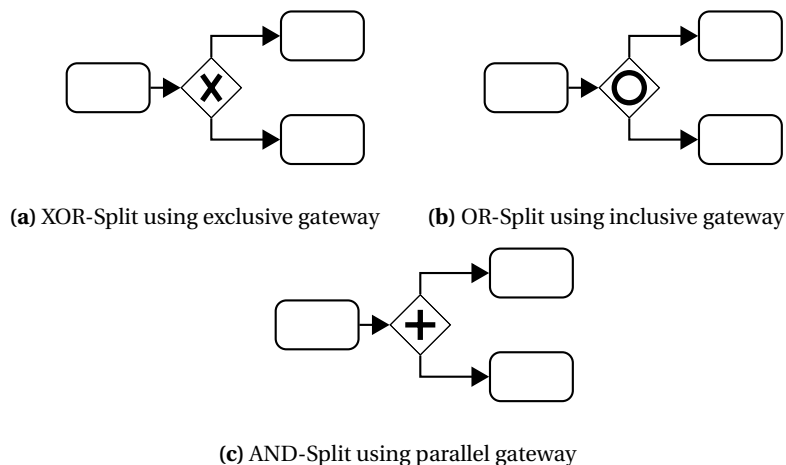
A consequence of the changes made so far in this section is that there is no longer a need for the special graphical notation used for conditional flows as shown in Figure 4.2 and described in [1, sec. 8.3.13]. This special depiction of conditional flows has to be used only if the source of the particular conditional flow is an activity.



---

**Figure 4.7** Splitting using gateways explicitly

---



---

In that case also at least one other outgoing sequence flow from that activity has to be present. Since activities can now have only one outgoing sequence flow and the mentioned depiction is not needed for outgoing conditional flows in gateways, this notation can be replaced using a gateway as shown further in this document. Functionality of modeling multiple outgoing conditional flows is still possible using gateways.

### 4.2.3 Tasks

The activity meta-model continues in [1, sec. 10.2.3] with tasks by defining seven distinct task types:

**Send task** designed to send a message,

**Receive task** designed to receive/wait for a message. Additionally, this type of task may instantiate a process,

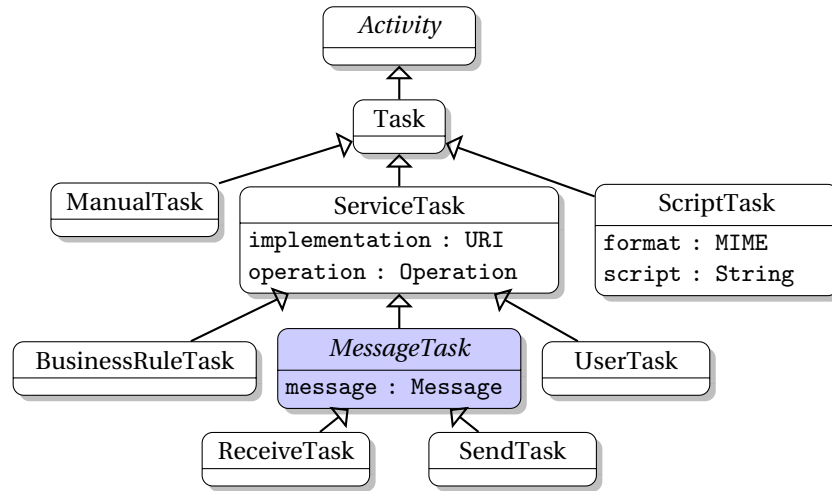
**Service task** using some kind of service,

**User task** designed to be performed by a human actor with the assistance of a supporting software,

**Manual task** also designed to be performed by a human actor, but with the difference to a user task that its life-cycle state is not managed by the Workflow Engine (WFE),

**Script task** designed to execute a script by the WFE, and

**Figure 4.8** Tasks meta-model class based on behavioral decomposition



**Business rule task** providing a mechanism to get calculations from a *Business Rule Engine*.

In the BPMN 2.0 specification these meta-model classes do not share any common ground, although they share some common behavior. We refine this meta-model class hierarchy in the sense shown in Figure 4.8. The arguments are the following: Both the ReceiveTask and the SendTask share common behavior, which can be inferred from [1, fig. 10.14] and we model it in Figure 4.8 as a common generalization with the *MessageTask* class. The ReceiveTask, the SendTask, the BusinessRuleTask and the UserTask share some kind of connection to operation implementation, which is first defined in the service task. Therefore, we use the ServiceTask as a common parent for the BusinessRuleTask, the UserTask and the new *MessageTask* classes. The BPMN says regarding the send task the following:

The send task has at most one `inputSet` and one data input. If the data input is present, it must have an item definition equivalent to the one defined by the associated message. At execution time, when the send task is executed, the data automatically moves from the data input on the send task into the message to be sent. If the data input is not present, the message will not be populated with data from the process. [1, p. 160]

Later the BPMN standard describes message intermediate events as follows:

A message intermediate event can either be used in normal control flow, similar to a send or receive task (for throw or catch message intermediate events, respectively), or it can be used in an event-based gateway [1, p. 456].

We keep send and receive tasks in the meta-model to support the BPMN graphical elements. Semantically they obviously do not intended to do anything significantly different than message events do. We compose this using the behaviors as discussed in section 5.2.1. Also note that in some related work the send and receive tasks are removed completely [44] since they are source of additional problems when used with event-based gateways or in case of instantiating receive tasks [58]. Due to the behavioral decomposition in this work those two elements will not result in more than two rules wrapping just the corresponding behaviors, e.g., the exclusive merge and parallel split behavior same as in intermediate events and activities, a service behavior as in service task, and a catch or throw behavior. This allows keeping the modeling elements without increasing the complexity of the ground model or introducing additional problems.

## 4.3 Events

In this section we will evaluate the event meta-model in [1] and propose enhancements improving the extensibility and structure of the original meta-model sub tree starting with the *Event* class.

The *Event* meta-model class is a direct descendant of the *FlowNode* class on the same level as the *Activity* and *Gateway* class [1, fig. 8.22] (see Figure 4.3). The BPMN 2.0 specification shows in the meta-model class diagram that this *Event* class has two direct descendants: *ThrowEvent* and *CatchEvent* [1, fig. 8.20] marked as group *A* in Figure 4.10. The three main event classes are *StartEvent*, *IntermediateEvent* and *EndEvent* [1, sec. 10.4] marked as group *B* in Figure 4.10. In the BPMN 2.0 meta-model [1, fig. 10.69] this is solved by separating intermediate events into *IntermediateThrowEvent* and *IntermediateCatchEvent* as two distinct meta-model classes, each having a different parent. The parent of the first is the *ThrowEvent* class, while the *CatchEvent* class is the parent of the latter. Still both classes share some common behavior or attributes, e.g., the amount of incoming/outgoing sequence flows. By separating intermediate events into distinct classes, such common behavior or attributes need to be duplicated. The overlapping behaviors of events as they appear in the BPMN 2.0 [1, fig. 8.20, sec. 10.4] is shown in Figure 4.9.

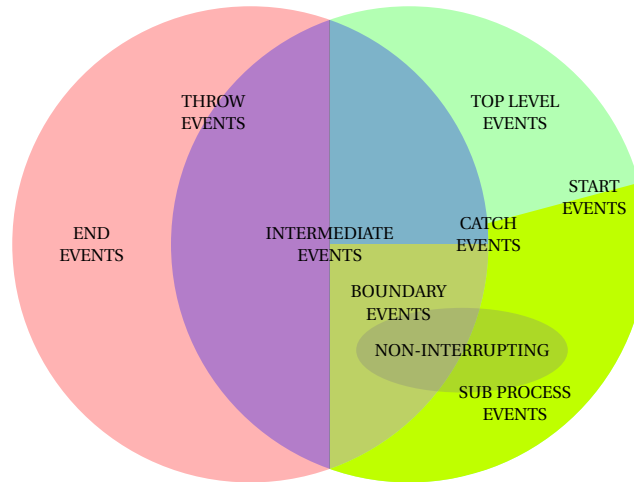
### 4.3.1 Events flow node

Targeting a high-level behavioral design, we need to identify why those classes are used in BPMN 2.0, what is the behavior they describe, what they share and what distinguishes them. Based on the event decomposition shown in Figure 4.9 the resulting class diagram may change from the original [1, fig. 8.20, fig. 10.69] to one in the sense shown in Figure 4.10. Newly inserted meta-model classes, originally not present in the BPMN 2.0 specification, are highlighted. The changes concern common attributes and behavior definition, and specific behavior separation. The common attributes and behavior definition is done by introducing the *IntermediateEvent* and *Interrupting* meta-model classes, which define the shared behavior for intermediate events or interrupting events respectively. The specific behavior sep-

---

**Figure 4.9** Decomposition of events based on their behavior

---

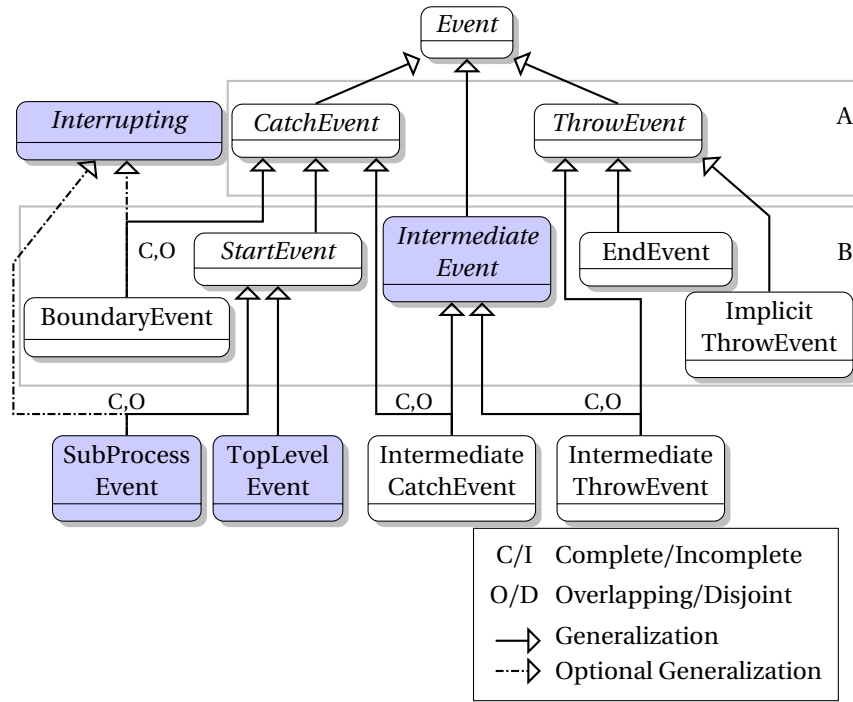


aration is carried out by the two new meta-model classes: *TopLevelEvent* and *SubProcessEvent*, and by the optional overlapping generalization of the *SubProcessEvent* and *BoundaryEvent* with the *Interrupting* meta-model class.

There are other ways to model this, but we argue the following. The *Event* meta-model class hierarchy starts with separating events into two abstract classes: the *CatchEvent* and the *ThrowEvent* like in BPMN 2.0 depicted as group A in Figure 4.10. On the next layer, depicted as group B in Figure 4.10, we specialize events to the abstract *StartEvent* class, the *EndEvent* class, the *BoundaryEvent* class, the *ImplicitThrowEvent* class and additionally into the abstract *IntermediateEvent* meta-model class. The slight difference between this and the BPMN 2.0 specification is that we introduced a common abstract *IntermediateEvent* class for both, the *IntermediateCatchEvent* class and the *IntermediateThrowEvent* class. Those two classes are then an overlapping specialization of the abstract *IntermediateEvent* class, and of the abstract *CatchEvent* class for the first, and of the abstract *ThrowEvent* class for the latter. We also introduce two new meta-model classes to specialize the abstract *StartEvent* class, namely the *TopLevelEvent* and the *SubProcessEvent* meta-model classes. Last the *Interrupting* class is an additional possible generalization of the *SubProcessEvent* or the *BoundaryEvent* class defining additional behavior.

While evaluating the BPMN meta-model some events may interrupt the encompassing sub-process in case of sub-process start events or cancel the activity they are attached to in case of boundary events. The first is modeled using *StartEvent#isInterrupting:boolean* attribute [1, tab. 10.87], while the latter is modeled using *BoundaryEvent#cancelActivity:boolean* [1, tab. 10.91]. Though both represent the same behavior they are represented by two distinct attributes in different classes. As interruption has to take some action to actually cancel the encompassing or attached activity, we model this using the common *In-*

**Figure 4.10** Event meta-model class based on behavioral decomposition



*interrupting* class representing such behavior. This class can be optionally inherited and the interrupting behavior injected in either the SubProcessEvent or the BoundaryEvent.

This meta-model then complies with the classification of events in [1, tab. 10.93], except for compensation triggers, which are in both cases referred to as interrupting events [1, tab. 10.93]. None of them, neither the sub-process start event nor the boundary event, actually interrupts the encompassing or attached activity, since such an activity is already completed when a compensation trigger occurs. Thus, we model the interrupting behavior as optional generalization rather than alternative as in the case of the abstract *IntermediateEvent*, the *IntermediateCatchEvent*, and the *IntermediateThrowEvent* classes, which would generate two extra classes and unnecessarily complicate the meta-model. This way we keep the meta-model as brief as possible at this level of abstraction. In the case of intermediate events, the refined meta-model still considerably matches the original one and the new or changed classes have a foundation in the BPMN 2.0 specification. This new meta-model class decomposition captures all possible event classes/types mentioned in the BPMN 2.0 specification [1]. We will deal with this refinement more in detail in chapter 5, where we will also discuss all the parameters and inherited behavior.

### 4.3.2 Events control flow

In this section the control flow related topics of events will be discussed. Unlike in section 4.2.2 the control flow will be discussed for start events, intermediate events, and end events separately in the following subsections.

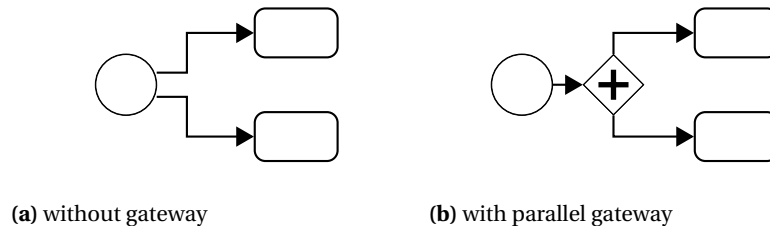
#### 4.3.2.1 Start events

In [1, p. 245] a start event must not be a target for sequence flows except for start events used in expanded sub-process. This is putting the use of the two new meta-model classes, namely the `TopLevelEvent` and the `SubProcessEvent`, where, e.g., the first will have no attribute for incoming any sequence flow, and the second will have an attribute for exactly one incoming sequence flow. Also a start event must be a source for at least one sequence flow. In the case a start event is a source for multiple sequence flows (see Figure 4.11a) they should be treated as parallel paths, where every such sequence flow has the parameter `conditionExpression` set to “None”<sup>1</sup>.

---

**Figure 4.11** Outgoing sequence flows from start events

---



Similarly as in case of activities, the option that a start event is a source of multiple sequence flows is avoided and replaced by using a parallel gateway as shown in Figure 4.11b. Whether to allow incoming sequence flows or not for sub-process start events. The case when a sub-process start event is used in an expanded sub-process is clear enough without using a sequence flow between the border of the sub-process and the start event. It is also in contradiction with the statement in [1, sec. 7.5.1] which says that if a sub-process is expanded within a diagram, the flow nodes within the sub-process cannot be connected to the objects outside of the sub-process. Connecting a start event to the border of the sub-process is in fact connecting it with the flow node, which is the source of the sequence flow, which is heading to the particular sub-process as said in [1, p. 245]. Nevertheless we leave this question for now and will discuss this issue further in chapter 5.

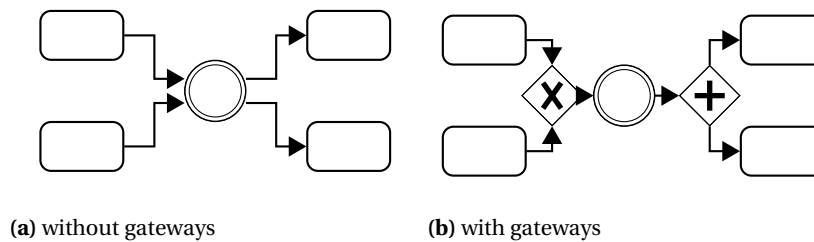
---

<sup>1</sup>Note that the attribute `conditionExpression` was already removed in section 4.1. This will be further refined in chapter 5 using conditions inside gateways

#### 4.3.2.2 Intermediate events

According to [1, p. 259] an intermediate event must be a target of one or more sequence flows and it must be also a source of one or more sequence flows. This behavior can be now defined/implemented in the added *IntermediateEvent* meta-model class and shared down the meta-model class hierarchy to both *IntermediateCatchEvent* and *IntermediateThrowEvent* classes.

**Figure 4.12** Incoming and outgoing sequence flows from and to an intermediate event

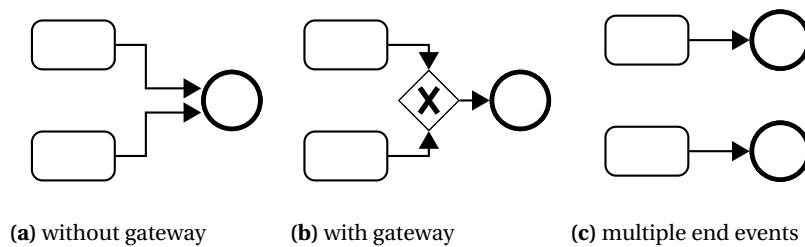


To follow the obvious intention of the above mentioned refinements, multiple incoming and/or outgoing sequence flows should be avoided and modeled using gateways as shown in Figure 4.12.

#### 4.3.2.3 End events

As described in [1, p. 249] an end event must be a target for a sequence flow, it may have also multiple incoming sequence flows and it must not have an outgoing sequence flow with the exception if the particular end event is used in an expanded sub-process.

**Figure 4.13** Incoming sequence flows to an end event



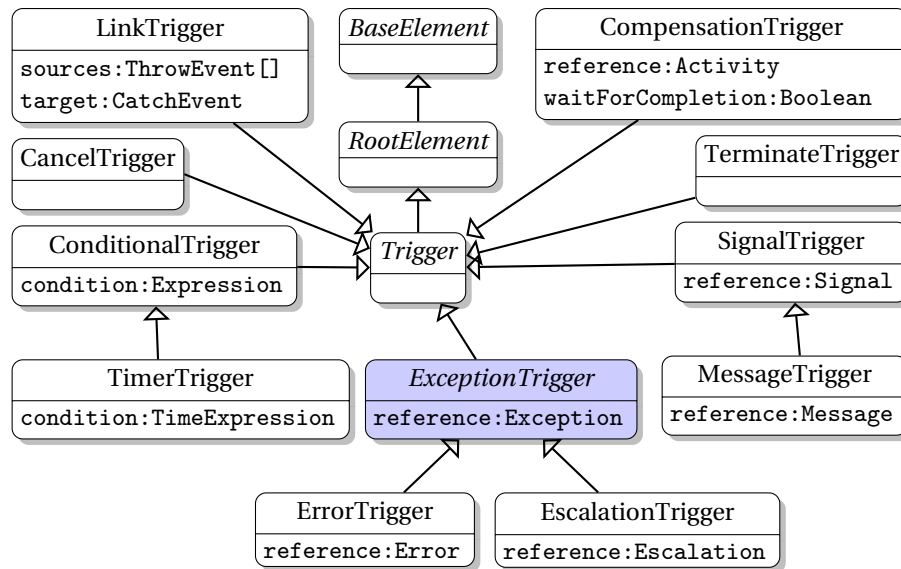
For the same reasons already mentioned in section 4.3.2.1 an end event can have only one incoming sequence flow. If there are more sequence flows heading to an

end event, it should be modeled using a gateway as shown in Figure 4.13b or using multiple end events as shown in Figure 4.13c and described in [1, p. 249].

### 4.3.3 Triggers

In the BPMN 2.0 specification the notion of “trigger” and “event definition” is overlapping and often confusing. Therefore, we use only “trigger” to define how an event can be set off. We will further avoid “event definition” in this thesis to eliminate confusion.

**Figure 4.14** Refactored trigger class hierarchy



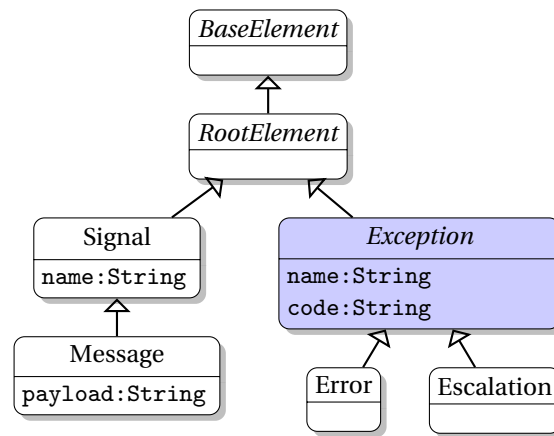
Triggers in the BPMN specification are: none triggers, message triggers, timer triggers, error triggers, escalation triggers, cancel triggers, compensation triggers, conditional triggers, link triggers, signal triggers, terminate triggers, multiple triggers, and parallel multiple triggers. The last two, the multiple triggers and the parallel multiple triggers, rather group other triggers than define a new trigger. Therefore we leave them out for now and will discuss them later. Although, the rest of the triggers are modeled in the BPMN 2.0 specification as distinct classes some common behavior can be observed.

In case of message triggers and signal triggers we know that both are forwarded by publication resolution and both are meant for inter-pool or inter-process communication. The first carries a message, while the second carries a signal and those differ only in the sense that message has a payload and signal does not [1, fig. 8.30, 10.73, 10.89, 10.93]. The major difference between messages and signals is that a message specifies a concrete target, i.e., can be caught only by one catch event, but



signal is broadcasted allowing to be caught by multiple catch events. Nevertheless the different behavior will be modeled on a lower level using the notification concept in chapter 5. We concentrate on the properties those two triggers have in common and model the MessageTrigger class as a descendant of the SignalTrigger, both of which having a reference, the first to a message and the second to a signal. Also since a message can be seen as an extension of signal by adding a payload (see Figure 4.15), the Message class is modeled as a descendant of the Signal class.

**Figure 4.15** Refactored signal, message, error and escalation class hierarchy



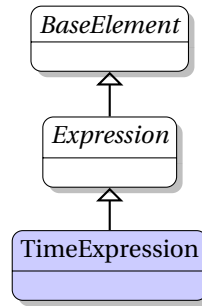
Both, the error trigger and the escalation trigger are meant for a situation when something unexpected happened during the execution of a process instance. The main difference is that an error is always interrupting but an escalation may not be. Since in this case those two trigger types have more in common than the previous pair, e.g., the forwarding mechanism including the targets, we model the ErrorTrigger and the EscalationTrigger with a common parent, the *ExceptionTrigger*. The new class is in Figure 4.14 highlighted as in previous figures. This way we also have the same decomposition for the two content elements, the Error class and the Escalation, which are both descendants of the common *Exception* class since both encompass a name and a code [1, fig. 10.73, 10.80, 10.82]. We still need the two classes, e.g., to prevent code collisions, since the code is used to match throw events and catch events [1, tab. 8.41, 8.42, 10.88]. Also the BPMN 2.0 specification is ambiguous in the means what event flow node type an error trigger or escalation trigger can be. While in [1, tab. 10.93] an error trigger can be defined for sub-process interrupting start event, boundary interrupting event and end event, [1, tab. 8.41] allows obviously also an intermediate event within the normal flow. We follow the definition in [1, tab. 10.93]. Based on the context in the BPMN 2.0 we came to the conclusion that the statement in [1, tab. 8.41] was incorrectly copied from [1, tab. 8.42]. Also the specification is not clear what mechanism is actually used to match corresponding error triggers and escalation triggers. In [1, tab. 8.41. 8.42] the code is defined for the

matching mechanism, while, e.g., [1, tab. 10.90] states that “a boundary error event reacts to (catches) a named error, or to any error if a name is not specified”. We use the matching technique based on code attribute for both, errors and escalations, since it seems the intention of this attribute [1, tab. 8.41, 8.42, 10.88]. We leave the name attribute for descriptive purposes [1, tab. 8.41, 8.42].

---

**Figure 4.16** Refactored expression class hierarchy

---



The last modification in the trigger meta-model comprises the conditional trigger and the timer trigger. The idea that both triggers are fired in case a kind of expression holds, is already deducible from [1, fig. 10.73]. A TimeExpression class as descendant of the Expression class makes it more general as defining a time condition using three distinct expressions.

The rest of the trigger classes, the CancelTrigger, the CompensationTrigger, the LinkTrigger and the TerminateTrigger remains as in the original BPMN 2.0 specification.

## 4.4 Gateways

The Gateway meta-model class is a direct descendant of the FlowNode class on the same level as the Activity and Event classes discussed in the previous sections (see Figure 4.3). The BPMN 2.0 specification shows in the meta-model class diagram that the Gateway class has five direct descendants (ExclusiveGateway, InclusiveGateway, ParallelGateway, ComplexGateway and EventBasedGateway), the gateway types [1, fig. 8.24]. The abstract Gateway class has one, a bit confusing, gatewayDirectionOfGateway attribute. The attribute represented by the static function gatewayDirection : GATEWAYS → GATEWAY\_DIRECTION can gain the following values [1, tab. 8.46]: *Unspecified*, *Converging*, *Diverging*, or *Mixed*. While a *Converging* gateway has to have at least one, but typically multiple, incoming sequence flows but exactly one outgoing sequence flow, and the other way around for a *Diverging* gateway - exactly one incoming sequence flow and one, but typically multiple, outgoing sequence flows, we argue that in case of exactly one incoming and exactly one outgoing sequence flow a gateway is neither a *Converging* nor *Diverging* gateway. We argue that the relevant values from the ones mentioned are

only *Converging*, *Diverging*, and *Mixed*. The *Unspecified* can then only occur in case of exactly one incoming and one outgoing sequence flow. This case we call *PassThrough*, which can be used to replace a conditional flow. Therefore we define the following:

**Pass through gateway** with exactly one incoming sequence flow and exactly one outgoing sequence flow (see Figure 4.17a). This is also replacing the conditional flow as shown in Figure 4.2.

**Merging gateway** with more than one incoming sequence flow and exactly one outgoing sequence flow (see Figure 4.17b),

**Splitting gateway** with exactly one incoming sequence flow and more than one outgoing sequence flows (see Figure 4.17c), and we leave the

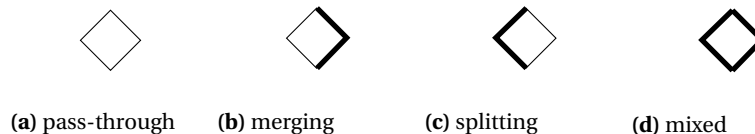
**Mixed gateway** with multiple incoming and multiple outgoing sequence flows (see Figure 4.17d).

We enhance the diamond depiction of gateways as shown in Figure 4.17 to explicitly show the gateway direction and improve readability of Process Diagrams (PDs). This graphical enhancement was inspired by the AND-split and AND-join building block [11, fig. 12]. The AND was seen as default, since it does not require any conditions on incoming nor outgoing edges - same as parallel gateway. This depiction enhancement is easily embeddable into the diamond depiction of a gateway [1, fig. 10.102] and it corresponds with the depiction of splitting or merging behavior further defined in chapter 6.

---

**Figure 4.17** Gateway direction

---



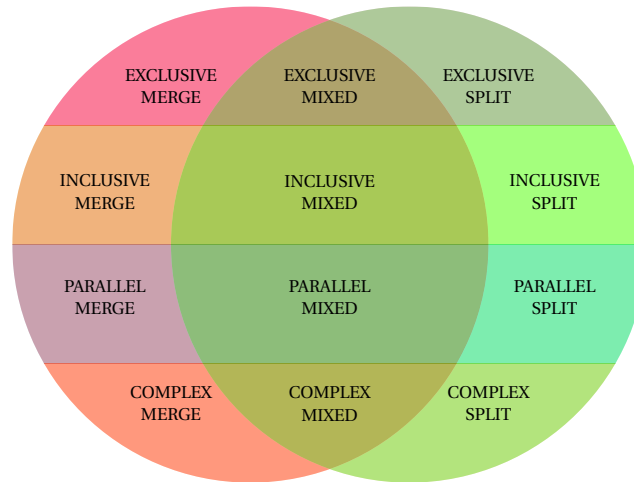
Gateways serve two main purposes. The first is to create conditional flow, which is represented in the simplest way by the *pass-through* gateway shown in Figure 4.17a and replaces the conditional flow modeling element shown in Figure 4.2. The second purpose of gateways is to merge or split the control flow [1, sec. 8.3.9]. Since a gateway always fits one of the above categories (see Figure 4.17), the default “Unspecified” value of the `gatewayDirection` attribute [1, tab. 8.46] is avoided. It also collides with the “Mixed” value at some point and causes confusion. There is also no default value for gateway direction at this moment and thus it needs to be always chosen resulting in a gateway related universe as shown in Figure 4.18. We can see that this universe is rather symmetric than the one shown for events in Figure 4.9.

The event-based gateways were not discussed right now, since they behave differently and will be discussed in a separate section 4.4.3 below.

---

**Figure 4.18** Decomposition of gateways based on their behavior

---

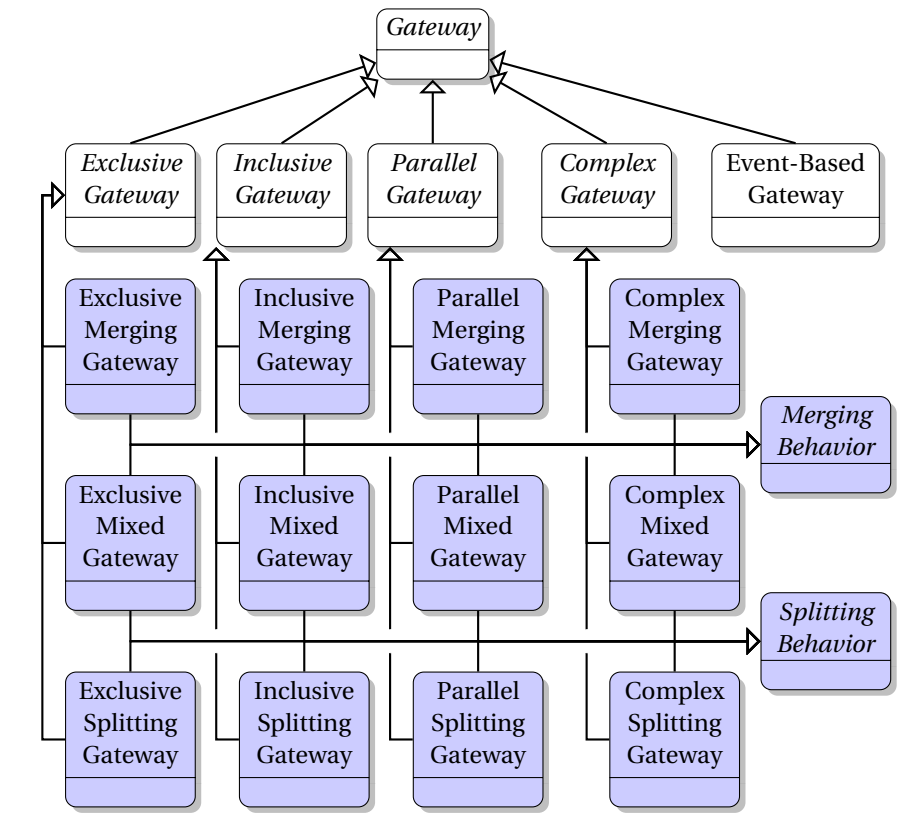


#### 4.4.1 Gateways flow node

Based on the gateway decomposition shown in Figure 4.18 the resulting class diagram may change in the sense shown in Figure 4.19. Newly inserted meta-model classes, originally not in the BPMN 2.0 specification, are highlighted. The class bloat apparent on the first look is not that drastic as it may appear. Only two meta-model classes are defining a behavior: the abstract *MergingGateway* class and the *SplittingGateway* class. The rest of the new classes just inherit the gateway type data-based behavior (*ExclusiveGateway*, *InclusiveGateway*, *ParallelGateway* or *ComplexGateway*) and inject one or both of the direction behaviors (see Figure 4.19). The reason of converting the *gatewayDirection* attribute into the meta-model classes is because of the behavioral modeling and will be further explained in chapter 5.

In [1, p. 288] an exclusive gateway can be depicted using both with or without the “X” marker (see Figure 4.20). Since this is also in contradiction with [1, p. 38], where merging is described as OR-Join, while the depiction shows an XOR-Join and this may lead — and obviously even in the BPMN 2.0 standard does — to confusion, we avoid the usage of the gateway depiction without any marker, which is also recommended by many others cp. [34, 33]. There is also no way with the current standard to draw an abstract diagram, where a gateway’s type may not be defined. Further in this document the gateway diamond symbol without any marker will be used exactly for such purposes, where the indication that the type of the gateway is *unspecified* (Figure 4.20a) is needed. A process containing such an *unspecified* gateway is *not executable*. Only the depiction with the “X” marker (Figure 4.20b) has to be used to explicitly model exclusive gateways. Therefore the gateway symbol without any marker can be used only in *abstract* diagrams, where it is not yet defined, what gateway will be used. Also no conditions to the outgoing sequence flows may be

**Figure 4.19** Gateways meta-model class based on behavioral decomposition



assigned for an *abstract* splitting gateway.

#### 4.4.2 Gateways control flow

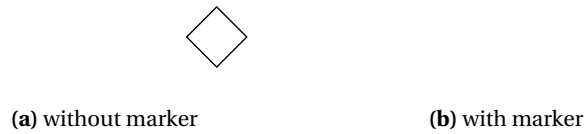
Data-based gateways may have one or more incoming and/or outgoing sequence flows. The configuration of the sequence flows depends on the `gatewayDirection` parameter. Allowed configurations of incoming sequence flows are:

- exactly one incoming sequence flow and exactly one outgoing sequence flow resulting in a *pass-through* gateway,
- exactly one incoming sequence flow and two or more outgoing sequence flows resulting in a splitting gateway,
- two or more incoming sequence flows and exactly 1 outgoing sequence flow resulting in a merging gateway, and

---

**Figure 4.20** Exclusive gateway depiction

---



- two or more incoming sequence flows and two or more outgoing sequence flows resulting in a mixed gateway.

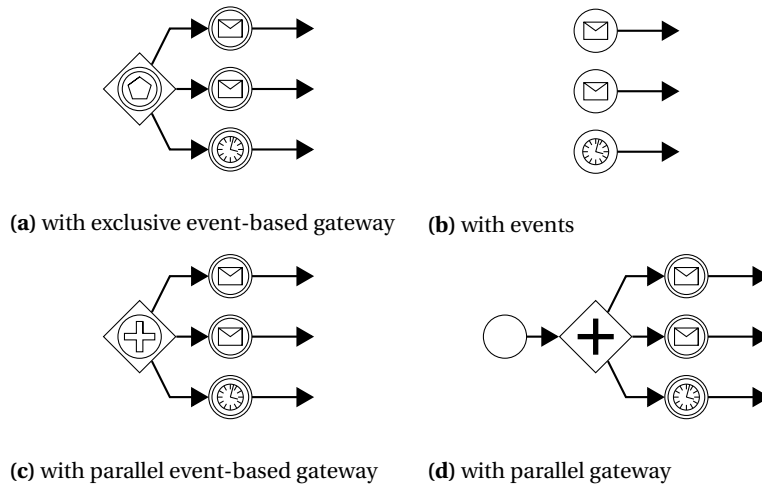
This means that every data-based gateway has to have at least one incoming and one outgoing sequence flow. It can have more than one incoming or outgoing sequence flows or both. A gateway has always to be classifiable as *pass-through*, splitting, merging or mixed. Special symbols should be used to differentiate between these gateway types. These symbols can be seen in Figure 4.17. A marker specifying the gateway type (inclusive (OR), exclusive (XOR), parallel (AND), complex (\*)) should be inserted inside the symbol in the same way as the BPMN 2.0 specification defines.

The option that a data-based gateway has no incoming or no outgoing sequence flow is not allowed withing the *basic* modeling elements. Using only *basic* modeling elements, this should be modeled always using a start or an end event respectively.

#### 4.4.3 Event-based gateway

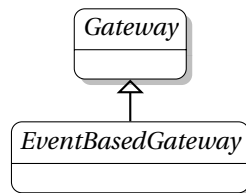
The event-based gateway concept in [1, sec. 10.5.6] represents a modeling mix between events and gateways. Compared to other flow nodes the event-based gateway configuration also contains all the flow elements targeted by its outgoing sequence flows. An event-based gateway can be either non-instantiating or instantiating. In case of non-instantiating, such a gateway is always exclusive represented by a race condition, where the first event that triggers the event-based gateway wins. In case of instantiating event-based gateway, such a gateway can be either exclusive or parallel. The difference between instantiating a process using an exclusive instantiating gateway (see Figure 4.21a) and using distinct start events (see Figure 4.21b) is that in the case the exclusive instantiating gateway is used the process instance already exists when waiting for one of the events to arrive. The parallel event-based gateway (see Figure 4.21c) then in addition does not disable (or maybe better: withdraws the functionality of disabling) the remaining events connected to the outgoing sequence flows of the gateway. The parallel event-based gateway starts a process instance, splits the control flow (generating a token for each outgoing sequence flow) and waits for all the target events to eventually arrive before the process instance can “normally” finish. This is semantically equivalent to starting a process instance using a start event and splitting the control flow using a parallel gateway as shown in Figure 4.21d. We see the parallel event-based gateway as redundant at this stage and therefore we remove it from the *basic* modeling elements. This allows us first

**Figure 4.21** Instantiation with event-based gateway vs. start events



to simplify the meta-model and remove the redundant `eventGatewayTypeOf-EventBasedGateway` attribute [1, tab. 10.127], which is ambiguously relevant only with a certain constraint, namely if the attribute `instatiateUsingEventBasedGateway` is true. Also, since the exclusive instantiating gateway is semantically equivalent to instantiating the process with a start event, splitting the control flow with a parallel gateway, where all parallel paths are targeting an intermediate event and waiting for all events to “happen”, we remove the `instatiateUsingEventBasedGateway` [1, tab. 10.127]. The simplified part of the meta-model regarding event-based gateways is shown in Figure 4.22

**Figure 4.22** Event-based gateway class hierarchy



## 4.5 Graphical representation

The used shapes, texts, colors, sizes, and lines for flow objects are inherited from the rules defined in [1]. Although some elements are enhanced or added, e.g., the

splitting and merging gateway, it is not contradicting or causing confusion with the graphical representation of the original BPMN flow elements.

The graphical representation of behaviors, e.g., merging behavior shown in Figure 6.3a and splitting shown in Figure 6.4, is not part of the graphical elements of the resulting modeling notation, but serves only to graphically describe the inner behavior of some meta-model objects. Still, also here the inherited BPMN symbols are kept.

In the next chapter we will formalize the modeling elements discussed in this chapter.



*My opinions may have changed, but not the fact that  
I am right.*

— Ashleigh Brilliant

## Chapter 5

# Enhanced Business Process Model and Notation (BPMN<sub>ε</sub>)

The purpose of this chapter is, first, to collect and point out ambiguities and inconsistencies in the BPMN 2.0 specification [1], propose corrections as well as enhancements and simplifications. Proposals in this chapter should not affect the modeling scope of BPMN 2.0 specification. Also the resulting models should still be backward compatible with the original BPMN 2.0 specification.

The idea is to start with a small-as-possible subset of the original BPMN 2.0 specifications and then refine this subset step-by-step by adding more elements and functionality, which will result in a specification, which is backward compatible with the original BPMN 2.0, but without any ambiguities and/or contradictions.

We will only concentrate on the parts of BPMN 2.0 specification which are within the focus of this thesis. Refinements of the rest of the BPMN 2.0 specification is proposed for future work in section 9.2

### 5.1 Universes and types

For the purpose of formal a description we use universes, which are based on set theory, on the higher level of abstraction, e.g., the business level. Targeting the implementation, this may be refined into a type system if needed. The usage of a type system may introduce unnecessary complexity [80], which is unwanted especially on higher levels of abstraction. As the BPMN meta-model already introduces a class hierarchy and an underlying type system we will first use this hierarchy in the manner of a set theory. We use the following standard universes:

`BOOLEANS` for logical values `true` and `false`,

`NUMBERS` for numbers,

`NATURAL_NUMBERS` for natural numbers  $\mathbb{N}$ ,

INTEGERS for integers  $\mathbb{Z}$ ,  
 RATIONAL\_NUMBERS for rational numbers  $\mathbb{Q}$ ,  
 REAL\_NUMBERS for real numbers  $\mathbb{R}$ ,  
 IMAGINARY\_NUMBERS for real numbers  $\mathbb{I}$ ,  
 COMPLEX\_NUMBERS for complex numbers  $\mathbb{C}$ ,  
 STRINGS for string values, etc.

Although not all of these universes may be used in the following ground model, they are defined here for the sake of completeness of the implementation of the ASM Ground Model  $\text{\LaTeX}$  package in Appendix C.

The set theory then allows us to build simple relationships between used universes, e.g., sub- and super universes:

$$\begin{aligned} \text{NATURAL\_NUMBERS} &\subset \text{INTEGERS} \subset \text{RATIONAL\_NUMBERS} \subset \\ &\subset \text{REAL\_NUMBERS} \subset \text{COMPLEX\_NUMBERS} \subset \text{NUMBERS} \end{aligned} \quad (5.1)$$

$$\text{IMAGINARY\_NUMBERS} \subset \text{COMPLEX\_NUMBERS} \quad (5.2)$$

$$\text{REAL\_NUMBERS} \cup \text{IMAGINARY\_NUMBERS} \in \text{COMPLEX\_NUMBERS} \quad (5.3)$$

The BPMN meta-model classes use the usual naming convention used in many languages, e.g., Java [90], Ruby [91], or C# [92]. Not to confuse class names with names of universes, the convention for universe names is upper case letters only, where in case of multiple words, those are separated by an underscore “\_” character. The name of a universe is always in plural, while the name of a class is always in singular. We assume, if not explicitly stated otherwise, that a camel case singular form representing a class corresponds with a plural uppercase, underscore separated form representing a universe, e.g.:

$$\text{FlowElement} \approx \text{FLOW\_ELEMENTS} \quad (5.4)$$

The BPMN meta-model generalization relationships [69] then define the following sub-/superset relations (see Figure 4.1 and 4.3):

$$\text{SEQUENCE\_FLOWS} \subset \text{FLOW\_ELEMENTS} \subset \text{BASE\_ELEMENTS} \quad (5.5)$$

$$\text{FLOW\_NODES} \subset \text{FLOW\_ELEMENTS} \subset \text{BASE\_ELEMENTS} \quad (5.6)$$

$$\text{SEQUENCE\_FLOWS} \cap \text{FLOW\_NODES} \equiv \emptyset \quad (5.7)$$

$$\text{ACTIVITIES} \subset \text{FLOW\_NODES} \quad (5.8)$$

$$\text{EVENTS} \subset \text{FLOW\_NODES} \quad (5.9)$$

$$\text{GATEWAYS} \subset \text{FLOW\_NODES} \quad (5.10)$$

$$\forall S, T \in \{\text{ACTIVITIES}, \text{EVENTS}, \text{GATEWAYS}\} (S \neq T) \rightarrow S \cap T \equiv \emptyset \quad (5.11)$$

### 5.1.1 Activity related universes

The following universes are based on Figure 4.8. The introduced enhancements in chapter 4 are incorporated in this section.

$$\text{TASKS} \subset \text{ACTIVITIES} \quad (5.12)$$

$$\text{MANUAL\_TASKS} \subset \text{TASKS} \quad (5.13)$$

$$\text{SERVICE\_TASKS} \subset \text{TASKS} \quad (5.14)$$

$$\text{SCRIPT\_TASKS} \subset \text{TASKS} \quad (5.15)$$

$$(5.16)$$

$$\forall S, T \in \{\text{MANUAL\_TASKS}, \text{SERVICE\_TASKS}, \text{SCRIPT\_TASKS}\} \quad (5.17)$$

$$(S \neq T) \rightarrow S \cap T \equiv \emptyset$$

$$\text{BUSINESS\_RULE\_TASKS} \subset \text{SERVICE\_TASKS} \subset \text{TASKS} \quad (5.18)$$

$$\text{MESSAGE\_TASKS} \subset \text{SERVICE\_TASKS} \subset \text{TASKS} \quad (5.19)$$

$$\text{USER\_TASKS} \subset \text{SERVICE\_TASKS} \subset \text{TASKS} \quad (5.20)$$

$$\forall S, T \in \{\text{BUSINESS\_RULE\_TASKS}, \text{MESSAGE\_TASKS}, \text{USER\_TASKS}\} \quad (5.21)$$

$$(S \neq T) \rightarrow S \cap T \equiv \emptyset$$

$$\text{RECEIVE\_TASKS} \subset \text{MESSAGE\_TASKS} \subset \text{TASKS} \quad (5.22)$$

$$\text{SEND\_TASKS} \subset \text{MESSAGE\_TASKS} \subset \text{TASKS} \quad (5.23)$$

$$\text{RECEIVE\_TASKS} \cap \text{SEND\_TASKS} \equiv \emptyset \quad (5.24)$$

### 5.1.2 Events related universes

Universes and their relationships regarding events including the enhancements from chapter 4, are incorporated in this section. The following universes are based on the meta-model class hierarchy shown in Figure 4.10:

$$\text{CATCH\_EVENTS} \subset \text{EVENTS} \quad (5.25)$$

$$\text{THROW\_EVENTS} \subset \text{EVENTS} \quad (5.26)$$

$$\text{START\_EVENTS} \subset \text{CATCH\_EVENTS} \subset \text{EVENTS} \quad (5.27)$$

$$\text{INTERMEDIATE\_EVENTS} \subset \text{EVENTS} \quad (5.28)$$

$$\text{END\_EVENTS} \subset \text{THROW\_EVENTS} \subset \text{EVENTS} \quad (5.29)$$

$$\text{CATCH\_EVENTS} \cap \text{THROW\_EVENTS} \equiv \emptyset \quad (5.30)$$

$$\text{BOUNDARY\_EVENTS} \subset \text{CATCH\_EVENTS} \subset \text{EVENTS} \quad (5.31)$$

$$\text{IMPLICIT\_THROW\_EVENTS} \subset \text{THROW\_EVENTS} \subset \text{EVENTS} \quad (5.32)$$

$$\begin{aligned} \forall S, T \in \{\text{START\_EVENTS}, \text{INTERMEDIATE\_EVENTS}, \text{END\_EVENTS}, \\ \text{BOUNDARY\_EVENTS}, \text{IMPLICIT\_THROW\_EVENTS}\} (S \neq T) \rightarrow S \cap T \equiv \emptyset \end{aligned} \quad (5.33)$$

$$\text{INTERMEDIATE\_CATCH\_EVENTS} = \text{CATCH\_EVENTS} \cap \text{INTERMEDIATE\_EVENTS} \quad (5.34)$$

$$\text{INTERMEDIATE\_THROW\_EVENTS} = \text{THROW\_EVENTS} \cap \text{INTERMEDIATE\_EVENTS} \quad (5.35)$$

$$\begin{aligned} \forall S, T \in \{\text{BOUNDARY\_EVENTS}, \text{START\_EVENTS}, \\ \text{INTERMEDIATE\_CATCH\_EVENTS}\} (S \neq T) \rightarrow S \cap T \equiv \emptyset \end{aligned} \quad (5.36)$$

$$\begin{aligned} \forall S, T \in \{\text{IMPLICIT\_THROW\_EVENTS}, \text{END\_EVENTS}, \\ \text{INTERMEDIATE\_THROW\_EVENTS}\} (S \neq T) \rightarrow S \cap T \equiv \emptyset \end{aligned} \quad (5.37)$$

$$\text{INTERMEDIATE\_CATCH\_EVENTS} \cap \text{INTERMEDIATE\_THROW\_EVENTS} \equiv \emptyset \quad (5.38)$$

$$\text{INTERMEDIATE\_CATCH\_EVENTS} \cap \text{START\_EVENTS} \equiv \emptyset \quad (5.39)$$

$$\text{INTERMEDIATE\_CATCH\_EVENTS} \cap \text{BOUNDARY\_EVENTS} \equiv \emptyset \quad (5.40)$$

$$\text{INTERMEDIATE\_THROW\_EVENTS} \cap \text{END\_EVENTS} \equiv \emptyset \quad (5.41)$$

$$\text{INTERMEDIATE\_THROW\_EVENTS} \cap \text{IMPLICIT\_THROW\_EVENTS} \equiv \emptyset \quad (5.42)$$

$$\text{TOP\_LEVEL\_EVENTS} \subset \text{START\_EVENTS} \quad (5.43)$$

$$\text{SUB\_PROCESS\_EVENTS} \subset \text{START\_EVENTS} \quad (5.44)$$

$$\text{TOP\_LEVEL\_EVENTS} \cap \text{SUB\_PROCESS\_EVENTS} \equiv \emptyset \quad (5.45)$$

Additionally the optional generalization of the *Interrupting* (see Figure 4.10) class of the BoundaryEvent class and the SubProcessEvent class assumes the following universes:

$$\begin{aligned} \text{INTERRUPTING\_BOUNDARY\_EVENTS} &\equiv \\ &\equiv \text{INTERRUPTING\_EVENTS} \cap \text{BOUNDARY\_EVENTS} \end{aligned} \quad (5.46)$$

$$\begin{aligned} \text{NON\_INTERRUPTING\_BOUNDARY\_EVENTS} &\equiv \\ &\equiv \text{BOUNDARY\_EVENTS} \setminus \text{INTERRUPTING\_BOUNDARY\_EVENTS} \end{aligned} \quad (5.47)$$

$$\begin{aligned} \text{INTERRUPTING\_TOP\_LEVEL\_EVENTS} &\equiv \\ &\equiv \text{INTERRUPTING\_EVENTS} \cap \text{TOP\_LEVEL\_EVENTS} \end{aligned} \quad (5.48)$$

$$\begin{aligned} \text{NON\_INTERRUPTING\_TOP\_LEVEL\_EVENTS} &\equiv \\ &\equiv \text{TOP\_LEVEL\_EVENTS} \setminus \text{INTERRUPTING\_TOP\_LEVEL\_EVENTS} \end{aligned} \quad (5.49)$$

### 5.1.3 Trigger related universes

Note that based on the refinements in section 4.3 the MessageTrigger is a subclass of the SignalTrigger. Consult Figure 4.14 for complete understanding of the defined relationships.

$$\text{TRIGGERS} \subset \text{ROOT\_ELEMENTS} \subset \text{BASE\_ELEMENTS} \quad (5.50)$$

$$\text{CANCEL\_TRIGGERS} \subset \text{TRIGGERS} \quad (5.51)$$

$$\text{COMPENSATION\_TRIGGERS} \subset \text{TRIGGERS} \quad (5.52)$$

$$\text{TIMER\_TRIGGERS} \subset \text{CONDITIONAL\_TRIGGERS} \subset \text{TRIGGERS} \quad (5.53)$$

$$\text{EXCEPTION\_TRIGGERS} \subset \text{TRIGGERS} \quad (5.54)$$

$$\text{LINK\_TRIGGERS} \subset \text{TRIGGERS} \quad (5.55)$$

$$\text{MESSAGE\_TRIGGERS} \subset \text{SIGNAL\_TRIGGERS} \subset \text{TRIGGERS} \quad (5.56)$$

$$\text{TERMINATE\_TRIGGERS} \subset \text{TRIGGERS} \quad (5.57)$$

$$\begin{aligned} \forall S, T \in \{\text{CANCEL\_TRIGGERS}, \text{COMPENSATION\_TRIGGERS}, \\ \text{CONDITIONAL\_TRIGGERS}, \text{EXCEPTION\_TRIGGERS}, \text{LINK\_TRIGGERS}, \\ \text{SIGNAL\_TRIGGERS}, \text{TERMINATE\_TRIGGERS}\} (S \neq T) \rightarrow S \cap T \equiv \emptyset \end{aligned} \quad (5.58)$$

$$\text{ERROR\_TRIGGERS} \subset \text{EXCEPTION\_TRIGGERS} \quad (5.59)$$

$$\text{ESCALATION\_TRIGGERS} \subset \text{EXCEPTION\_TRIGGERS} \quad (5.60)$$

Some triggers define certain payload to carry additional information. Payload may be also needed to match throw events with the corresponding catch events. Those payload types were shown in Figure 4.15 and result into the following universes:

$$\text{MESSAGES} \subset \text{SIGNALS} \subset \text{ROOT\_ELEMENTS} \subset \text{BASE\_ELEMENTS} \quad (5.61)$$

$$\text{EXCEPTIONS} \subset \text{ROOT\_ELEMENTS} \subset \text{BASE\_ELEMENTS} \quad (5.62)$$

$$\text{ERRORS} \subset \text{EXCEPTIONS} \quad (5.63)$$

$$\text{ESCALATIONS} \subset \text{EXCEPTIONS} \quad (5.64)$$

$$\text{ERRORS} \cap \text{ESCALATIONS} \equiv \emptyset \quad (5.65)$$

Additionally since TimerTrigger became a specialization of ConditionalTrigger (see Figure 4.14), a new TimeExpression was added into the meta-model (see Figure 4.16) resulting in the following universes:

$$\text{TIME\_EXPRESSIONS} \subset \text{EXPRESSIONS} \subset \text{BASE\_ELEMENTS} \quad (5.66)$$

#### 5.1.4 Gateways related universes

Universes based on Figure 4.19 and 4.22 are shown in this section. We also infer *MergingGateway* and *SplittingGateway*, which are depicted as behavior classes (*MergingBehavior* and *SplittingBehavior*) in Figure 4.19. This was originally done with the purpose that both, merging and splitting behaviors, are not only gateway related.

$$\text{EXCLUSIVE\_GATEWAYS} \subset \text{GATEWAYS} \quad (5.67)$$

$$\text{INCLUSIVE\_GATEWAYS} \subset \text{GATEWAYS} \quad (5.68)$$

$$\text{PARALLEL\_GATEWAYS} \subset \text{GATEWAYS} \quad (5.69)$$

$$\text{COMPLEX\_GATEWAYS} \subset \text{GATEWAYS} \quad (5.70)$$

$$\text{EVENT\_BASED\_GATEWAYS} \subset \text{GATEWAYS} \quad (5.71)$$

$$\text{MERGING\_GATEWAYS} \subset \text{GATEWAYS} \quad (5.72)$$

$$\text{SPLITTING\_GATEWAYS} \subset \text{GATEWAYS} \quad (5.73)$$

$$\text{MERGING\_GATEWAYS} \cap \text{SPLITTING\_GATEWAYS} \equiv \emptyset \quad (5.74)$$

$$\begin{aligned} \forall S, T \in \{\text{EXCLUSIVE\_GATEWAYS}, \text{INCLUSIVE\_GATEWAYS}, \\ \text{PARALLEL\_GATEWAYS}, \text{COMPLEX\_GATEWAYS}, \\ \text{EVENT\_BASED\_GATEWAYS}\} (S \neq T) \rightarrow S \cap T \equiv \emptyset \end{aligned} \quad (5.75)$$

$$\begin{aligned} \text{EXCLUSIVE\_MERGING\_GATEWAYS} &\equiv \\ &\equiv \text{EXCLUSIVE\_GATEWAYS} \cap \text{MERGING\_GATEWAYS} \end{aligned} \quad (5.76)$$

$$\begin{aligned} \text{INCLUSIVE\_MERGING\_GATEWAYS} &\equiv \\ &\equiv \text{INCLUSIVE\_GATEWAYS} \cap \text{MERGING\_GATEWAYS} \end{aligned} \quad (5.77)$$

$$\begin{aligned} \text{PARALLEL\_MERGING\_GATEWAYS} &\equiv \\ &\equiv \text{PARALLEL\_GATEWAYS} \cap \text{MERGING\_GATEWAYS} \end{aligned} \quad (5.78)$$

$$\begin{aligned} \text{COMPLEX\_MERGING\_GATEWAYS} &\equiv \\ &\equiv \text{COMPLEX\_GATEWAYS} \cap \text{MERGING\_GATEWAYS} \end{aligned} \quad (5.79)$$

$$\begin{aligned} \text{EXCLUSIVE\_SPLITTING\_GATEWAYS} &\equiv \\ &\equiv \text{EXCLUSIVE\_GATEWAYS} \cap \text{SPLITTING\_GATEWAYS} \end{aligned} \quad (5.80)$$

$$\begin{aligned} \text{INCLUSIVE\_SPLITTING\_GATEWAYS} &\equiv & (5.81) \\ &\equiv \text{INCLUSIVE\_GATEWAYS} \cap \text{SPLITTING\_GATEWAYS} \end{aligned}$$

$$\begin{aligned} \text{PARALLEL\_SPLITTING\_GATEWAYS} &\equiv & (5.82) \\ &\equiv \text{PARALLEL\_GATEWAYS} \cap \text{SPLITTING\_GATEWAYS} \end{aligned}$$

$$\begin{aligned} \text{COMPLEX\_SPLITTING\_GATEWAYS} &\equiv & (5.83) \\ &\equiv \text{COMPLEX\_GATEWAYS} \cap \text{SPLITTING\_GATEWAYS} \end{aligned}$$

$$\begin{aligned} \text{EXCLUSIVE\_MIXED\_GATEWAYS} &\equiv & (5.84) \\ &\equiv \text{EXCLUSIVE\_GATEWAYS} \cap \text{MERGING\_GATEWAYS} \cap \text{SPLITTING\_GATEWAYS} \end{aligned}$$

$$\begin{aligned} \text{INCLUSIVE\_MIXED\_GATEWAYS} &\equiv & (5.85) \\ &\equiv \text{INCLUSIVE\_GATEWAYS} \cap \text{MERGING\_GATEWAYS} \cap \text{SPLITTING\_GATEWAYS} \end{aligned}$$

$$\begin{aligned} \text{PARALLEL\_MIXED\_GATEWAYS} &\equiv & (5.86) \\ &\equiv \text{PARALLEL\_GATEWAYS} \cap \text{MERGING\_GATEWAYS} \cap \text{SPLITTING\_GATEWAYS} \end{aligned}$$

$$\begin{aligned} \text{COMPLEX\_MIXED\_GATEWAYS} &\equiv & (5.87) \\ &\equiv \text{COMPLEX\_GATEWAYS} \cap \text{MERGING\_GATEWAYS} \cap \text{SPLITTING\_GATEWAYS} \end{aligned}$$

## 5.2 Meta-model

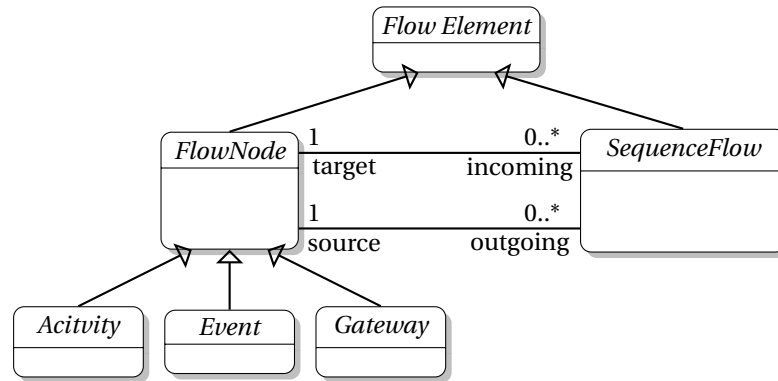
The goal of the BPMN<sub>ε</sub> is to clarify and fix the BPMN meta-model in a deeper sense than done in the previous chapters by even adding new, yet unspecified, parts and defining the *compound elements* using the *basic elements* from chapter 4. This is done separately for each modeling element and other parts of the standard. Regarding the flow node meta-model class hierarchy we use the basic decomposition shown in Figure 4.3.

We base the class hierarchy on the one present in [1] and formalized in [17]. The goal is to enable reuse of defined elements and simplify the model. The basic decomposition of diagram contents can be seen in Figure 5.1. We intentionally left out the optional `isImmediate` attribute for the sequence flow class since this is interpreted as `true` if no value is provided and must not be `false` for executable processes [1, tab. 8.51].

### 5.2.1 Tasks

The task meta-model was refined as shown in Figure 4.8. Nevertheless the send task and receive task somehow seem to duplicate the functionality of a message event.

**Figure 5.1** Basic decomposition of diagram contents



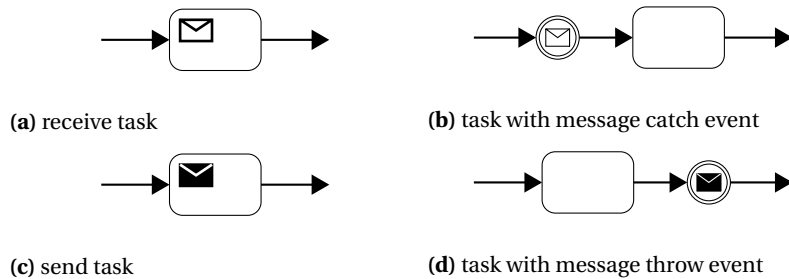
The only difference is that the send task or receive task are both *simple* tasks invoking a service operation, while the message event does not. This is semantically equivalent to prepending a message event in case of a receive task or appending one in case of a send task (see Figure 5.2). We base the sequence of flow nodes in Figure 5.2d on the assumption that a throw event is an atomic operation, but a service task is not and may even throw an error or escalation. Therefore the throw event is placed after the task. A different situation is in the case depicted in Figure 5.2b. Here we need to wait for the event first and then start any service operation. Both depictions show how a send task or receive task will be composed from the behavior point of view and in the case shown in Figure 5.2d the `lifeCycleStateOf-ActivityInInstance` will be set to “Completed” after the throw event. Reusing the behaviors from those two *basic elements* we can model the BPMN 2.0 receive and send task. Additionally, the constraint in event-based gateways that it cannot target a mix of receive tasks and message catch events [1, sec. 10.5.6] is void by this definition, since it always targets an event. If a receive task should be used to start a process, then in Figure 5.2b a message start event should be used instead of the depicted message intermediate event.

### 5.2.2 Gateways

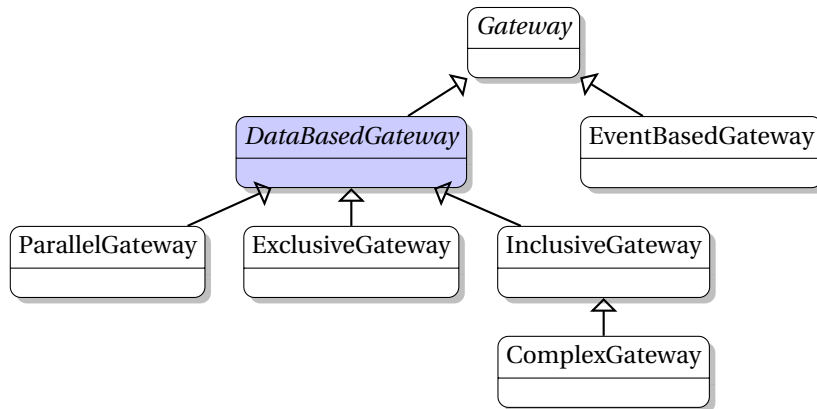
Using a similar behavior based decomposition approach for gateways as for events resulted in a gateway subclass bloat as shown in Figure 4.19. Therefore, we model the penultimate classes from the class tree shown in Figure 4.18 as behaviors in chapter 6 and build the concrete gateway as a combination of the defined behaviors. This will also enable those gateway related behaviors to be reused in other flow nodes and model extended modeling elements, i.e., splitting, merging, inclusive decision without a gateway or conditional flow [1, tab. 7.2]. We refactor the gateway class hierarchy as shown in Figure 5.3. First we separate data-based gateways from event-based gateways, which is the essential difference between the two major gate-



**Figure 5.2** Receive task and send task vs. task with message event



**Figure 5.3** Refactored gateway class hierarchy



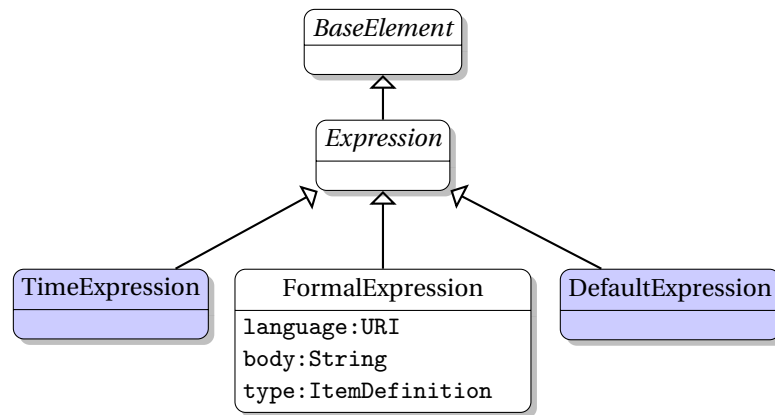
way types, and move the **ParallelGateway**, **ExclusiveGateway** and **InclusiveGateway** meta-model classes as subclasses of the new **DataBasedGateway** class. Next, we identify similarity between the inclusive gateway and the complex gateway. The complex gateway is seen here as a specialization of the inclusive gateway. The firing part of both, the complex gateway and the inclusive gateway, behaves the same way by evaluating conditions on all outgoing conditional flows in any order, producing a token on all of them, which evaluate to `true` [1, tab. 13.3, 13.5]. The activation part is a bit more complex for the complex gateway. The complex gateway introduces a state with the `waitingForStartOfComplexGateway` attribute, where this state is initially `true` and becomes `false` as soon as the `activationConditionExpressionForComplexGateway` evaluates to `true` [1, tab. 10.125]. The complex gateway fires and waits for reset behaving similarly as the inclusive gateway waiting to be fired, but ignoring all incoming sequence flows which contributed to start the complex gateway [1, tab. 13.5]. Resetting the complex gateway results in firing it again and setting the `waitingForStartOfComplexGateway` at-

tribute back to `true`. The resulting meta-model class hierarchy can be seen in Figure 5.3.

---

**Figure 5.4** Refined expression class hierarchy

---



In order to endorse reusability and to simplify the meta-model we do avoid defining the default flow as a new sequence flow type and rather define a default expression (see Figure 5.4), a constant, which will be further used in defining the default path for gateways in case no other conditional flow holds.

### 5.2.3 Events, triggers, notifications

In the BPMN 2.0 specification an event is something that “happens” during a process run [1] at an unpredictable point of time. The term “event” is also used in the BPMN specification to denote an event flow node which is waiting for an event (something to happen). The definition of something to “happen” in order to fire the particular event flow node is defined using a trigger or also an event definition. In this document we will further use the following naming conventions for those terms to avoid confusions but still preserve the context familiar with BPMN:

**Event or event node** identifying a particular flow node type from the BPMN meta-model used in a PD.

**Trigger** an event definition defining what has to “happen” in order for the corresponding event node to fire. The term event definition will not be further used in this document to prevent ambiguity. A trigger is associated with an event node defining its type.

**Notification** is a materialization of “something happened”, represented by a notification object defining what “happened” and additional information, e.g., timestamp or payload. Such an object has to reach an event node so it can fire

similarly as tokens do in sequence flows. Not all triggers have a corresponding notification, only those whose forwarding technique requires them<sup>1</sup>.

A notification is sometimes called differently in the literature, e.g., *trigger* or *event object*, which may lead to confusion. We use the three definitions above in this thesis to avoid confusion with each other. Since event is already used for event node we avoid the usage of the word “event” in the other two namings. Therefore, we explicitly use trigger for an event definition and notification for an event trigger as an object in the event flow similar as a token in the control flow.

The separation of triggers and notifications leads to a change in the original BPMN 2.0 meta-model [1, fig. 10.73], where triggers are represented by classes such as: *CancelEventDefinition*, *CompensateEventDefinition*, *ConditionalEventDefinition*, *ErrorEventDefinition*, *EscalationEventDefinition*, *LinkEventDefinition*, *MessageEventDefinition*, *SignalEventDefinition*, *TerminateEventDefinition* and *TimerEventDefinition* (see Figure 4.14 for the refined meta-model). Notifications are then represented by classes such as: *Error*, *Escalation*, *Message* and *Signal* (see Figure 4.15 for the refined meta-model). Since in the original meta-model a notification is hard bounded to a trigger, such a notification has to be defined twice for throw and catch event with the corresponding trigger. Also in case of a message trigger the payload is coming with the trigger, even if such a payload will generally not be yet defined. Therefore, we detach triggers from notifications, resulting in the changes shown in Figure 4.14 and Figure 5.5. Note that the meta-model class hierarchy shown in Figure 5.5 is a refinement of the class hierarchy shown in Figure 4.15 and replaces it.

The specification distinguishes implicit and explicit triggers. Implicit triggers such as conditional, timer or link trigger are resolved directly. Explicit triggers can be classified by the way they are resolved. There are five ways of how a trigger can be resolved [1, sec. 10.4.1]:

**Publication resolution** [1, sec. 10.4.1] is used for triggers having a corresponding notification and allows the notification to be caught by any catch event in any scope. This is used for message and signal triggers communicating between pools and processes.

**Propagation resolution** [1, sec. 10.4.1] is also used for triggers having a corresponding notification, which is forwarded from the location it was triggered to the innermost enclosing scope instance up the process instance hierarchy tree. Such an enclosing scope is generally represented by an activity with an attached boundary event able to catch the notification. If no such catch event is found and the notification reaches the root of the process instance tree, such a notification will remain unresolved. It is then up to the WFE how such a situation will be handled (see Part III for details about WFE).

**Compensation resolution** [1, sec. 10.4.1] is bounded with a concrete activity and instance, it triggers a compensation handler of the activity in the instance in

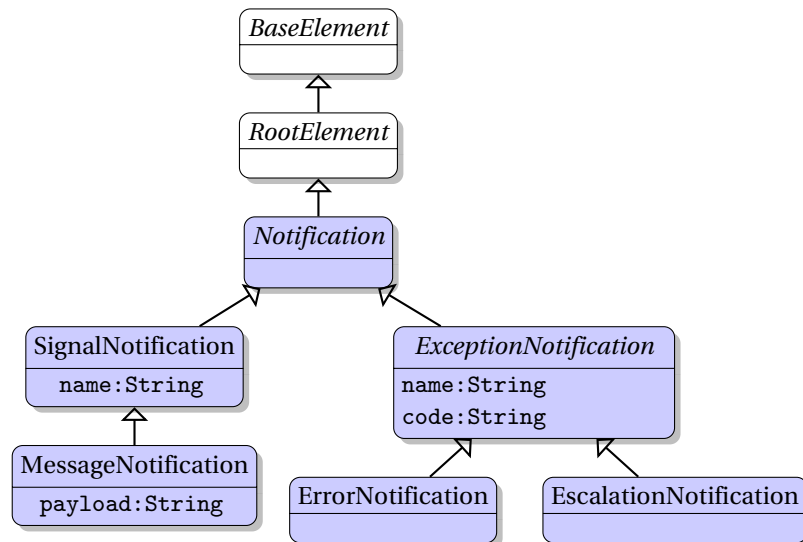
---

<sup>1</sup>The notification concept is not needed on the business level of abstraction and will be introduced formally on the technical level in Part III

---

**Figure 5.5** Notifications class hierarchy

---



which it was fired. Compensation triggers do not have a corresponding notification object.

**Cancellation resolution** [1, sec. 10.4.1] is bounded with a concrete activity and instance. It triggers a compensation handler of all completed activities inside the canceled activity and terminates all running activities. If the canceled activity is a transaction sub-process, it will be rolled back. Cancel triggers do not have a corresponding notification object.

**Termination resolution** [1, sec. 10.4.1] is bounded with a top-level process instance, which should be terminated, meaning that all running activities inside such a process and instance should be ended without any compensation or any other handling. Terminate triggers do not have a corresponding notification object.

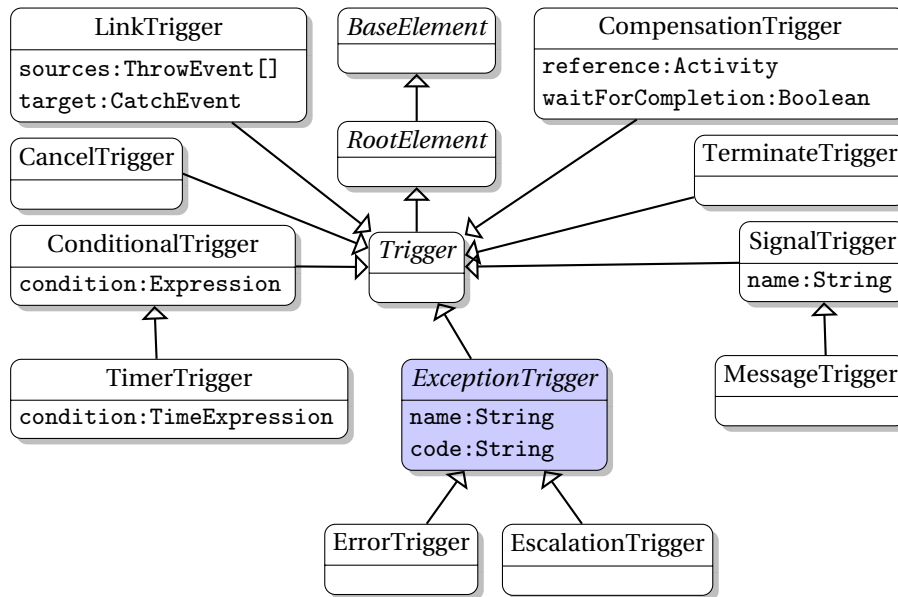
Message and signal notifications are very similar. They are both forwarded by publication and both are meant for inter-pool or inter-process communication. They differ mainly in the fact that message triggers are defined as pairs, one throw event and one catch event, while multiple catch events may define the same signal triggers for one throw event. Another difference between message and signal notifications is that signal notifications do not have a payload, while message notifications do. Matching of corresponding message triggers is realized in BPMN using a message flow. But this is not possible for executable processes since two different communicating processes may be running on two different machines. Therefore we use the same matching technique as for signal triggers using the `nameOfSignal-`

Notification attribute [1, fig. 8.30] defined also in the corresponding trigger. Because of the above we see a message trigger as an extension of signal trigger. Due to their similarities we change the structure of the meta-model hierarchy as shown in Figure 4.14, where the message trigger class is a descendant of the signal trigger class and similarly for notifications shown in Figure 5.5.

Similarities may be also observed between the escalation trigger and the error trigger in Figure 4.14 and in the corresponding notifications in Figure 5.5. The corresponding notification classes are both introducing a `codeOfExceptionNotification` for the escalation and error notification respectively and use a similar matching technique as signal notification or message notification. They differ from message notification or signal notification in the direction they are forwarded (see propagation resolution forwarding above). We reflect those similarities in the common abstract *ExceptionTrigger* class for triggers in the meta-model (see Figure 4.14) and in the common *Notification* class for notifications (see Figure 5.5).

A timer trigger is seen as a special case of a conditional trigger. They share the structure and differ in the type of the expression to be evaluated in order to fire such implicit trigger. This is also reflected in the *Expression* meta-model class hierarchy shown in Figure 5.4.

**Figure 5.6** Refactored trigger class hierarchy removing links between triggers and notifications



There is no link between the notification and the trigger since notifications do not exist before they are thrown and matched to a concrete event node using a certain trigger. This has been done by refactoring the class hierarchy shown in Fig-

ure 4.14 into the one in Figure 5.6. This results in removing attributes, such as: `errorReferenceOfSignalTrigger` from error trigger, `messageReferenceOfSignalTrigger` from message trigger, `escalationReferenceOfSignalTrigger` from escalation trigger or `signalReferenceOfSignalTrigger` from signal trigger and moving attributes such as `nameOfExceptionNotification` or `codeOfExceptionNotification` from error trigger and escalation trigger. The `nameOfSignalNotification` attribute is moved to `SignalTrigger` and the `nameOfExceptionNotification` is moved to *ExceptionNotification* and inherited by all their descendants (`MessageTrigger`, `EscalationTrigger` and `ErrorTrigger`). The `codeOfExceptionNotification` attribute is moved to *ExceptionTrigger* and inherited by `EscalationTrigger` and `ErrorTrigger`.

*Beware of bugs in the above code; I have only proved  
it correct, not tried it.*

— Donald Knuth

## Chapter 6

# Formalization of behaviors

During the work on formalizing the BPMN meta-model duplicates appeared to be the reason for many of the inconsistencies in the existing BPMN standard. Thus, we try to extract the common behavior patterns to avoid such cases resulting in a revised, more compact and graspable meta-model. This work is based on Process Diagrams (PDs) as described in BPMN 2.0 specification. This particular notation is chosen since it is popular between process designers and modelers and it is the most commonly used graphical representation for business processes [76]. We believe that the usability of our outputs will be supported by the fact that the potential reader does not have to work him/herself into a completely new process modeling notation.

Our refinements follow the process execution conformance [1, sec. 2.2] and tries to formalize the particular part, we call behaviors using the ASM method [39, 2] as this method is used in this work as a formalization tool.

We start with the BPMN modeling practices [33, 34, 35, 36, 37], which contain among others gateway pairing or restriction of sequence flows on one or both flow node sides (incoming/outgoing side of a flow node). This may improve readability of simple diagrams, increase clearness and reduce ambiguity of even complex diagrams on one hand but increase complexity of the diagrams in general since the amount of flow nodes in a PD dramatically increases on the other hand. By simplifying the meta-model we may achieve an unambiguous, clear, correct and more graspable meta-model, which is important for the conformance and deployment. But the complexity growth of such process models may be harder to maintain during development and lower sustainability of processes based on such meta-model.

In this chapter we will show the formalized BPMN 2.0 specification focusing on the control flow. Also as discussed in section 1.5 the approach is to decompose the BPMN 2.0 standard and the defined flow elements into behavior patterns. In this section the basic behavior patterns will be formally defined. Those patterns will be reused to specify the concrete flow elements. We define behavior patterns, similar to [17], and then reuse such patterns as building blocks to define the BPMN elements. This has, among others, the advantage that the intermediate step of defining *simple flow nodes* as discussed in section 1.5 is not necessary. Also we can dynami-

cally switch those building blocks using for example polymorphic or other suitable techniques. This allows us to define one element instead of multiple container elements forming all possible combinations. E.g., the activity example in Figure 1.3 would result in more than one such container if we would also consider activities with no incoming or outgoing sequence flows or any combinations of the above. An example of a parallel gateway behavior decomposition is shown in Figure 1.4.

This is the core decomposition idea we will use in this work to formally model the BPMN elements. This introduction should sketch the abilities and advantages of behavioral decomposition. Also this goes in hand with the requirements capture of a system [2], which is BPMN in our case, as we tend to describe the behavior of each modeled element and split them into small parts based on similarities and observation.

For depicting behaviors we use rectangles with sharp corners. This was chosen not to collide with any BPMN symbol. Inside such a rectangle resides a BPMN or a custom symbol identifying the behavior. We use the designation “simple” for flow elements which deal only with the core of their purpose. E.g., an activity behavior does not deal with multiple incoming or outgoing sequence flows, just with performing an activity - dealing with the activity life-cycle. Similarly, does the simple gate behavior deal with just the decision making, e.g., if a token may or may not pass to a sequence flow. Or a simple gateway behavior, which deals with merging or splitting the control flow. The quantity of incoming and outgoing sequence flows is done by the input or output behavior which is shown in Figure 1.4 and Figure 1.5 as join and split behavior. In the next sections we will show how this can be implemented.

As in the BPMN 2.0 specification the basic flow nodes — activities, events and gateways — are described in the alphabetic order. In this chapter we will exceptionally start with gateways since their behavior patterns are essential for the rest of the flow nodes.

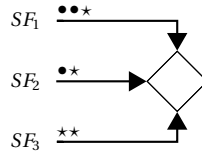
This chapter starts with the common *enabling token sets* concept in section 6.1 used in all flow node types to determine the activation. In section 6.2 the gate behavior, used to determine which outgoing sequence flows should be taken for all flow node types and also reused to define conditional flow, is described. In section 6.3 and 6.4 the behaviors for merging and splitting the control flow will be discussed resulting in the definition of gateway transition rules in 6.5 as an example. An example on horizontal extension will be shown in section 6.6, where the original BPMN specification will be enriched by a sole exclusive gateway. In section 6.7 the event behavior will be introduced.

## 6.1 Enabling token sets

The `enablingTokenSetsOfFlowNode` returns a set of such token sets. In every token set all tokens have to belong to one instance. No two tokens residing in the same sequence flow can be present in one token set. For example, in Figure 6.1 there is a flow node with three `incomingSequenceFlowsOfFlowNode`  $SF_1$ ,  $SF_2$  and  $SF_3$  on which tokens from two different instances,  $I_A$  and  $I_B$ , resides. A token from instance  $I_A$  is depicted as “•” and a token from instance  $I_B$  is depicted as “★”. The token dis-



**Figure 6.1** Enabling token sets



tribution is the following: sequence flow  $SF_1$  contains two tokens of instance  $I_A$  and one token of instance  $I_B$ , sequence flow  $SF_2$  contains one token of each instance and sequence flow  $SF_3$  contains two tokens of instance  $I_B$ . Thus the following four sets of tokens (see Table 6.1) are returned: Set 1 will contain two tokens of instance  $I_A$  from sequence flow  $SF_1$  and  $SF_2$ . Set 2 will contain one token of instance  $I_A$  from sequence flow  $SF_1$ . Set 3 will contain three tokens of instance  $I_B$  from all the three incoming-SequenceFlowsOfFlowNode. And last, set 4 will contain one token of instance  $I_B$  from the sequence flow  $SF_3$ . Each of the token sets may or may not fire the given flow node depending on the `ActivationBehaviorOfFlowNode` defining the abstract behavior for merging possibly multiple sequence flows of a flow node. This rule returns all relevant instances containing the given flow node in which the flow node should fire. The size of the set identifies the number of fires and the process instances in which to fire. The tokens, which contributed to any of the firing and are supposed to be consumed, are returned by the abstract `firingTokensOfFlowNode` and the process instances in which to fire the flow node are returned using the abstract `firingInstancesOfFlowNode`.

**Table 6.1** Enabling token sets

Token set	1	2	3	4
$SF_1$	•	•	•	
$SF_2$	•		•	
$SF_3$			•	•

## 6.2 Gate behavior

A gate is a mechanism, which somehow controls the flow. As in the real world a gate has a guard watching the gate and letting in only those who fulfill certain conditions modeled here by the `gateConditionForSequenceFlow` in listing 7.5.

A simple gate can be seen as a condition, which has to hold if a token `canPassThroughFlowNodeUsingSequenceFlowInInstanceWithGateBehaviour`. This mechanism will be further used in controlled flow, which is realized in the BPMN 2.0 specification using conditional flows and gateways. Since all

of those flow elements share this behavior, it is extracted here as a building block intended to be reused further in this work. To notate this behavior, a small diamond shape shown in Figure 6.2 is used as this is shared for both the conditional flows and gateways [1].

---

**Figure 6.2** Gate behavior symbol

---



The example of a gate can be explained on a real life example from the Middle Ages:

A fortress has one or more gates. There is usually a main gate and then there might be one or more side gates or back gates. Each of these gates has a guard group represented by one or more soldiers. Within one fortress those guards have always a way to communicate with guards at other gates inside the same fortress. There is also a kind of law or a leader who decides how the gates get opened or closed.

The BPMN representation of the above example is the following: a fortress is a flow node, a guard is a `gateConditionForSequenceFlow` and the law is the `GATE_BEHAVIOR` — e.g., “PARALLEL”, “EXCLUSIVE”, or “INCLUSIVE”. A gate basically defines if a token `canPassThroughFlowNodeUsingSequenceFlowInInstanceWithGateBehaviour`. This depends on the mentioned `gateConditionForSequenceFlow` and on the `GATE_BEHAVIOR` (see Equation 6.1). Using the “INCLUSIVE” `GATE_BEHAVIOR`: a token `canPassThroughFlowNodeUsingSequenceFlowInInstanceWithGateBehaviour` if the `gateConditionForSequenceFlow` holds. The second, “EXCLUSIVE” `GATE_BEHAVIOR` behaves as follows: a token `canPassThroughFlowNodeUsingSequenceFlowInInstanceWithGateBehaviour` if the `gateConditionForSequenceFlow` holds and for no other sequence flow the `gateConditionForSequenceFlow` holds. A special handling is done for `DEFAULT_EXPRESSIONS` as `gateConditionForSequenceFlow`. This will hold in the case when no other sequence flow the `gateConditionForSequenceFlow` holds, no matter if the `GATE_BEHAVIOR` is “EXCLUSIVE” or “INCLUSIVE”. A “PARALLEL” `GATE_BEHAVIOR` behaves a bit different. It lets a token pass with all the other sequence flows at once.

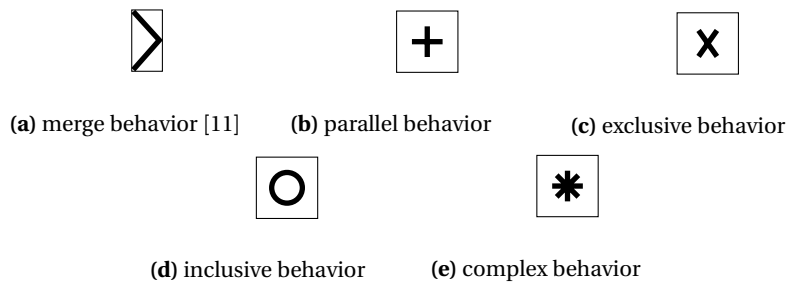
$$\text{GATE\_BEHAVIOR} \in \{ \text{“INCLUSIVE”}, \text{“EXCLUSIVE”}, \text{“PARALLEL”} \} \quad (6.1)$$

### 6.3 Merge behavior

A flow node may have one or more `incomingSequenceFlowsOfFlowNode`. In case of multiple `incomingSequenceFlowsOfFlowNode` such as a flow node needs

some kind of `MergeBehaviorOfFlowNode`. For a `MergeBehaviorOfFlowNode` the symbols shown in Figure 6.3a [11] in conjunction with the different types - `ParallelMergeBehaviorOfFlowNode` shown in Figure 6.3b, `ExclusiveMergeBehaviorOfFlowNode` shown in Figure 6.3c, `InclusiveMergeBehaviorOfFlowNode` shown in Figure 6.3d, and `ComplexMergeBehaviorOfFlowNode` shown in Figure 6.3e. A square is used to identify a behavior in order not to be confused with any BPMN flow element. For the different `MergeBehaviorOfFlowNode` types, we use the same symbol as in BPMN 2.0, inside the square symbolizing a behavior.

**Figure 6.3** Merge behaviors



A `MergeBehaviorOfFlowNode` in general identifies possible `enabling-TokenSetsOfFlowNode`, which are able to fire the given flow node. Every such set may fire the given flow node once or even multiple times in some cases. A token set in this sense is a combination of tokens residing on distinct directly `incoming-SequenceFlowsOfFlowNode` of the given flow node with a distribution possibly able to fire the given flow node. There are no tokens in one set, which share the same incoming sequence flow and instance of a process or sub-process. A set of such sets will be used further in this document to handle all possible, even multiple fires across the process or sub-process instances. Every time a flow node fires, all the `firingTokensOfFlowNode` will be consumed. Such `firingTokensOfFlowNode` are a union of all tokens in all `firingTokensOfFlowNode`, which can actually fire the given flow node. The `MergeBehaviorOfFlowNode` defines the abstract behavior pattern for merging multiple `incomingSequenceFlowsOfFlowNode`. This rule returns all relevant instances of a (sub-)process containing the given flow node in which the flow node fired. The size of the set of these `firingInstancesOfFlowNode` identifies the number of fires and each item of this set is the process instance in which the given flow node fired. Any tokens which contributed to any of the fires returned by the abstract `firingTokensOfFlowNode` function has to be in one of the instances returned by the `firingInstancesOfFlowNode` function and has to be consumed.

There are four types of `MergeBehaviorOfFlowNode` defined in the BPMN 2.0 specification:

`ParallelMergeBehaviorOfFlowNode` firing the flow node if there is a token on all `incomingSequenceFlowsOfFlowNode` [1, tab. 13.1]. Possible multiple instances have to be taken into account. This is realized by the mentioned `enablingTokenSetsOfFlowNode`, where each token set can contain only tokens from the same instance. The `ParallelMergeBehaviorOfFlowNode` has to check then if each set contains a token for every directly incoming sequence flow.

`ExclusiveMergeBehaviorOfFlowNode` firing the flow node if there is at least one token on at least one incoming sequence flow of that flow node [1, tab. 13.2]. Taking multiple instances into account, this behavior will fire the given flow node for every token residing in any directly incoming sequence flow in any instance.

`InclusiveMergeBehaviorOfFlowNode` firing the flow node if there is at least one token on at least one incoming sequence flow and there are no `UpstreamTokensOfFlowNode` related to that flow node [1, tab. 13.3].

`ComplexMergeBehaviorOfFlowNode` defining two activation behaviors. First is when the flow node is in `waitingForStartOfComplexGateway` and second if it is not in that state. In the first case the flow node gets fired if the `activationConditionExpressionForComplexGateway` evaluates to true and in the second case if all upstream tokens from the first phrase arrive [1, tab. 13.5], i.e., the same as `InclusiveMergeBehaviorOfFlowNode` without `sequenceFlowsToIgnoreDuringResetOfComplexGateway`, which are those fired the flow node in the `waitingForStartOfComplexGateway` state.

The set of `firingTokensOfFlowNode` are only those, which contribute to fire one of the `firingInstancesOfFlowNode`. All other tokens residing on any of the directly `incomingSequenceFlowsOfFlowNode` will not be included in that set.

## 6.4 Split behavior

Flow nodes may also have one or more `outgoingSequenceFlowsFromFlowNode`. Such a flow node then needs some kind of `SplitBehaviorOfFlowNodeAndInstance`. For a `SplitBehaviorOfFlowNodeAndInstance` the symbols shown in Figure 6.4 [11] in conjunction with the different types - `ParallelSplitBehaviorOfFlowNodeInInstance` shown in Figure 6.3b, `ExclusiveSplitBehaviorOfFlowNodeInInstance` shown in Figure 6.3c, and `InclusiveSplitBehaviorOfFlowNodeInInstance` shown in Figure 6.3d. The `SplitBehaviorOfFlowNodeAndInstance` defines the abstract behavior for splitting the flow into multiple `outgoingSequenceFlowsFromFlowNode`. This rule produces a token on all `allowedOutgoingSequenceFlowsFromFlowNode` for the given flow node.

---

**Figure 6.4** Split behavior [11]

---



---

In general, `SplitBehaviorOfFlowNodeAndInstance` performs `ProduceTokenOnSequenceFlowForInstance` for every `allowedOutgoingSequenceFlowsFromFlowNode`. According to the BPMN 2.0 specification we distinguish three different split behaviors:

`ParallelSplitBehaviorOfFlowNodeInInstance` defining all `outgoingSequenceFlowsFromFlowNode` as `allowedOutgoingSequenceFlowsFromFlowNode` and producing a token on all `outgoingSequenceFlowsFromFlowNode`.

`ExclusiveSplitBehaviorOfFlowNodeInInstance` determining if a token `canPassThroughFlowNodeUsingSequenceFlowInInstanceWithGateBehaviour`, where the used `GATE_BEHAVIOR` is “EXCLUSIVE”, and producing a token on exactly one `outgoingSequenceFlowsFromFlowNode`, where the `gateConditionForSequenceFlow` allows such a passage.

`InclusiveSplitBehaviorOfFlowNodeInInstance` determining if a token `canPassThroughFlowNodeUsingSequenceFlowInInstanceWithGateBehaviour`, where the used `GATE_BEHAVIOR` is “INCLUSIVE”, and producing a token on any subset of `outgoingSequenceFlowsFromFlowNode`, where the `gateConditionForSequenceFlow` allows such a passage.

## 6.5 Gateway transitions

Now the required building blocks for gateway are available (we discuss here only data-based gateways, see Figure 5.3). The `GatewayTransition` can be built along with all the specific gateway type transition rule. The `DataBasedGatewayTransition` gathers relevant `enablingTokenSets` and if the concrete `MergeBehaviorOfFlowNode` allows firing the given flow-based gateway, it will produce tokens on all `allowedOutgoingSequenceFlowsFromFlowNode`.

There are as many gateway types as there are `MergeBehaviorOfFlowNode` types. The number of different gateway and `SplitBehaviorOfFlowNodeAndInstance` types only differ in one, as the inclusive and the complex gateway share the same `SplitBehaviorOfFlowNodeAndInstance`, namely the `InclusiveSplitBehaviorOfFlowNodeInInstance`.

The `ParallelGatewayTransition` shown in listing 7.43, the `ExclusiveGatewayTransition` in listing 7.44, and the `InclusiveGatewayTransition` in listing 7.45 are then simply composing the existing behavior patterns. This will be

that simple for other possible `FlowNodeTransition` rules as well, e.g., for activities or events. Other flow nodes will of course refine a different rule than the `DataBasedGatewayTransition` used for data-based gateways, where their specific behavior and additional constraints may be defined.

This defines the `ParallelGatewayTransition` as `DataBasedGatewayTransition` using the `ParallelMergeBehaviorOfFlowNode` as `MergeBehavior` and the `ParallelSplitBehaviorOfFlowNodeInInstance` as `SplitBehaviorOfFlowNodeAndInstance`. Similarly for `ExclusiveGatewayTransition` and `InclusiveGatewayTransition` by using the `ExclusiveMergeBehaviorOfFlowNode` as `MergeBehaviorOfFlowNode` and the `ExclusiveSplitBehaviorOfFlowNodeInInstance` as `SplitBehaviorOfFlowNodeAndInstance` for the first one, and the `InclusiveMergeBehaviorOfFlowNode` as `MergeBehaviorOfFlowNode` and `InclusiveSplitBehaviorOfFlowNodeInInstance` as `SplitBehaviorOfFlowNodeAndInstance` for the later. For `ComplexGatewayTransition` a special `ComplexMergeBehaviorOfFlowNode` as `MergeBehaviorOfFlowNode` is used, but as `SplitBehaviorOfFlowNodeAndInstance` the `InclusiveSplitBehaviorOfFlowNodeInInstance` is reused.

## 6.6 Sole exclusive gateway

We will demonstrate how simple it is to extend the ground model, sketched in this thesis of a different behaviors and flow node types. Let's look closer at the exclusive gateway. What should happen, if there are more than one token distributed on any of the incoming sequence flows. We defined here that each of these tokens will fire the given gateway, resulting in possible multiple fires in one instance and step. This behavior is based on the BPMN 2.0 specification [1], e.g., the exclusive merging behavior for exclusive gateways and activities. But this might be also seen as a possible ambiguity. In some cases we might want to use the exclusive gateway in situations, where more than one token is present on any distinct incoming sequence flow is not permitted. There is no way to do that with the currently existing flow node types and behaviors defined earlier. Therefore, we decided to extend the ground model.

If a gateway is allowed to have only one incoming sequence flow containing a token the rule `MutuallyExclusiveMergeBehavior` shown in listing 7.12 should be used. Thus, more than one incoming sequence flow containing a token is considered as a violation of this rule and will raise an `MultipleTokensAtMutuallyExclusiveMerge` error. On the other hand multiple tokens in the same incoming sequence flow will not raise such an error. This is considered as correct behavior and will fire the given flow node for each token residing in such a sequence flow.

We do not discuss here if more tokens on the same incoming sequence flow is a correct interpretation of an exclusive gateway or not. We allow this since more than one token can be produced on the same sequence flow for example if the `completionQuantityOfActivity` attribute is greater than 1 [1, tab. 10.3]. The goal of sole exclusive gateway is to correctly merge alternative paths - distinct incoming sequence flows to a given flow node.

The goal of this example is to show that if such a behavior needs clarification, it can be easily introduced by extending the ground model. We leave the discussion about correct behavior of an exclusive gateway to some future work.

## 6.7 Event behavior

In this section we will provide the underlying formal semantics for the enhanced meta-model described in section 4.3. Events are used in BPMN to model when something has “happened” during the run of a process instance. The *Event-Transition* defines the semantics of the *Event*. It refines the *Workflow-Transition* [16] as shown in listing 7.31. This transition rule will *SelectFiringInstancesOfEvent* and activate if there is at least one selected firing instance with performing a event type specific *ControlOperationOfEvent* and *Event-OperationOfEvent*. Those operations will be further refined in event type specific sub-rules. Here we deviate from the work in [16], where we split the control and event specific conditions and operations only for events, since all other flow node types do not define neither event specific condition nor operation. The selection of firing instances is an intersection of both, firing instances based on *EventActivationBehaviorOfEvent* and firing instances based on *Control-ActivationBehaviorOfEvent*. If only one of those two behaviors is defined then only that one is used. In case both of those behaviors are defined then the resulting firing instances from each of these behaviors are cached until their counterpart occurs.

On the first level of the *Event* meta-model class hierarchy (see group *A* in Figure 4.10) we find the *CatchEvent* class represented by *CatchEventTransition* shown in listing 7.33, and the *ThrowEvent* class represented by the *ThrowEventTransition* shown in listing 7.34. Both of these rules refine only the event related parts of *EventTransition* and none of the control flow ones.

On the next level of the *Event* meta-model class hierarchy (see group *B* in Figure 4.10) is the event typification to start events, intermediate events, end events, and boundary events. These are represented by the rules *StartEventTransition* (see listing 7.35), *IntermediateEventTransition* (see listing 7.37), *EndEventTransition* (see listing 7.36), and *BoundaryEventTransition* (see listing 7.40) respectively. The rules *StartEventTransition* and *BoundaryEventTransition* have similar behavior in the sense that they both define only *EventActivationBehaviorOfEvent* and they define a similar *Control-OperationOfEvent* by using the *ParallelSplitBehaviorOfFlowNodeIn-Instance* where the *StartEventTransition* will additionally *CreateInstance* for the encompassing activity and set the firing instance as the parent instance of the new instance. In both cases, the start event and the boundary event may be a source of multiple *outgoingSequenceFlowsFromFlowNode* forming parallel paths realized with the *ParallelSplitBehavior* discussed in section 6.4.

Implicit throw events are not defined on this level of abstraction since they do not have a corresponding graphical element. This type will be needed on lower levels, where the Workflow Interpreter (WFI) will be in focus (see Part III).

All these rules related to event meta-model classes depicted as group *B* in Figure 4.10 define the control related activation behavior and operation. The `ControlActivationBehaviorOfEvent` returns a set of instances for which the given event should fire, where this behavior will consider incoming tokens. In case of a start event, shown in listing 7.35, there are no `incomingSequenceFlowsOfFlowNode` [1, sec. 10.4.2] and therefore the corresponding transition will consider only event occurrence defined by the refined `CatchEventTransition` (see listing 7.33). In case of an end event, shown in listing 7.36 the event reacts only on the incoming tokens reusing the `ExclusiveMergeBehavior` [77, fig. 9].

More complex behavior is defined by the `IntermediateEventTransition` shown in listing 7.37. In this case the event needs to wait for arriving tokens on the incoming sequence flows and additionally for the occurrence of the corresponding events in case of an `IntermediateCatchEvent`. We model this as follows: as soon as a token arrives, its instance will be stored in a temporary `waiting` location and the token will be consumed. Similarly for occurring events. There are some instances `waiting` for additional constraints to be fulfilled and can be selected as firing instances of the event. This is further refined in the `IntermediateCatchEventTransition` and the `IntermediateThrowEventTransition`, where the first waits for an event to occur, and the later does not.

In the next chapter we will transit the defined behaviors and transitions to the technical level and define the discussed behavior and functionality using a formal pseudo code. Listings from the next chapter were already referenced from this chapter, which was possible thanks to the ASM Ground Model  $\mathbb{M}\mathbb{E}\mathbb{X}$  package (see Appendix C). This package represents a support tool developed during writing this thesis, which allows us to define a link between the formal objects on the business level used in this chapter and the functions and rules in chapter 7.



*The pure and simple truth is rarely pure and never simple.*

— Oscar Wilde

## Chapter 7

# The ground model ( $BPMN_{GM}$ )

Now we will apply the 4-step transition approach defined in section 3.3 and transform the formal objects from the business level of our specification to the technical level. With business level we mean the part of this work described in all previous chapters of Part II, namely chapter 4 Modeling elements in BPMN, where the basic modeling elements are extracted and the BPMN meta-model is refined. Then chapter 5 Enhanced Business Process Model and Notation, where the refined meta-model is formalized and extended. The complete list of transitions can be looked up in Appendix A. Last, chapter 6 Formalization of behaviors, where behaviors are defined. We followed the mentioned extraction of formal objects as described in chapter 2 and 3. Those formal objects are easily extracted since they are defined with the support of the *ASMGM*  $\LaTeX$  package (see Appendix C).

Then the signatures of those formal objects are transformed using a parser defined in listing 3.3 with additional universe aliases defined in listing 7.1. Next step is to manually implement the bodies of the rules and the derived functions based on the description and meta-model from chapter 4, 5 and 6. The resulting ground model is then shown in this chapter starting with general functions and rules in section 7.1. Section 7.2 shows the behaviors including, gate behavior, activation behavior, and output behavior. Next, the flow node related functions and rules will be described in section 7.3, followed by functions and rules related to each flow node type, thus, activities in section 7.4, events in section 7.5, and gateways in section 7.6. Last, the removed parts of the original BPMN specification discussed in previous chapters will be enumerated in section 7.7.

---

**Listing 7.1** Universe aliases for parsing formal objects from the business level of abstraction

---

```

1  # Functions
2  /(can|is)[A-Z][a-z]+/ → "BOOLEAN"
3  /[a-z]+Occurred/ → "BOOLEAN"
4  /(completed|interrupted|instantiate)/ → "BOOLEAN"
5  /waitingForStart/ → "BOOLEAN"
6  /[a-z]+([A-Z][a-z]+) Expression/ → "EXPRESSIONS"
7  /[a-z]+FlowNode/ → "FLOW_NODES"
8  /activeInstances/ → "SET(INSTANCES)"
9  /[a-z]+Instances/ → "MULTISET(INSTANCES)"
10 /[a-z]+([A-Z][a-z]+) SequenceFlow/ → "SEQUENCE_FLOWS"
11 /[a-z]+([A-Z][a-z]+) SequenceFlows/ → "SET(SEQUENCE_FLOWS)"
12 /sequenceFlows([A-Z][a-z]+) / → "SET(SEQUENCE_FLOWS)"
13 /[a-z]+TokenSets/ → "SET(SET(TOKENS))"
14 /signalReference/ → "SIGNALS"
15 /messageReference/ → "MESSAGES"
16 /exceptionReference/ → "EXCEPTIONS"
17 /errorReference/ → "ERRORS"
18 /escalationReference/ → "ESCALATIONS"
19
20 # Rules with return values
21 /(. Input|Activation|. +Merge) Behavior/ → "MULTISET(INSTANCES)"
22
23 # Rules with no return values
24 /(Sole)?(Input|Output) Behavior/ → VOID
25 /(Parallel|(Mutually)?Exclusive|Inclusive|Complex)?MergeBehavior/ → VOID
26 /(Parallel|Exclusive|Inclusive)?SplitBehavior/ → VOID
27 /((Start)?EventBehavior)/ → VOID
28 /Split|Throw/ → VOID
29 /([A-Z][a-z]+)+Transition/ → VOID

```

---

## 7.1 General functions and rules

In this section the general functions and rules are placed. These functions and rules usually define a global property or behavior and are used across the ground model.

The first two static functions, which can be seen in signature 7.1 and 7.2, hold the source flow node or target flow node respectively of sequence flow as defined in [1, tab. 8.51].

```
static sourceFlowNode : SEQUENCE_FLOWS → FLOW_NODES [7.1]
```

```
static targetFlowNode : SEQUENCE_FLOWS → FLOW_NODES [7.2]
```

Tokens, which are meant to traverse through the diagram, have to belong to an instance. This is defined by the controlled function `instance` shown in signature 7.3.

```
controlled instance : TOKENS → INSTANCES [7.3]
```

The derived function `enablingTokenSets : FLOW_NODES → SET(SET(TOKENS))` shown in listing 7.2 returns sets of set of tokens. In every token set all tokens have to belong to one instance and no two tokens residing in the same sequence flow can be present in the same token set. See section 6.1 for detailed information about enabling token sets.

Since a set of tokens belonging to the same instance is used often in the ground model, e.g., enabling token sets above, we define a `instance` derived function, which takes a set of tokens and assuming, they are all from one instance it returns such instance by picking one random token from the given set and returning its instance.

The derived function `evaluate` shown in signature 7.4 evaluates the given expression in the given instance and returns the result of the evaluation. The own evaluation depends on the implementation in the WFI and the `EXPRESSION` language has to be supported by the WFI.

```
abstract derived evaluate : EXPRESSIONS × INSTANCES →  
BOOLEAN [7.4]
```

The rule `WorkflowTransition` shown in listing 7.4 handles both, the transition between the static flow nodes in the process definition and the transition between the life-cycle states of all instances of the given flow node [44].

---

**Listing 7.2** derived enablingTokenSets : FLOW\_NODES → SET(SET(TOKENS))

---

```
1 derived enablingTokenSets(flowNode) =  
2 return res in  
3   local selectedToken, allTokens, tokenSet ← ∅ in  
4     res ← ∅  
5     allTokens ← { token | token ∈ TOKENS  
6                 ∧ sequenceFlow(token) ∈ incomingSequenceFlows(flowNode) }  
7  
8     while allTokens ≠ ∅ do  
9       selectedToken ← undef  
10  
11     if tokenSet = ∅ then  
12       choose token ∈ allTokens do selectedToken ← token  
13     else  
14       choose token ∈ allTokens :  
15         ∀ t ∈ tokenSet with  
16           sequenceFlow(t) ≠ sequenceFlow(token)  
17           ∧ instance(t) = instance(token) holds do  
18         selectedToken ← token  
19  
20     if selectedToken ≠ undef then  
21       add selectedToken to tokenSet  
22       remove selectedToken from allTokens  
23  
24     if selectedToken = undef ∨ allTokens = ∅ then  
25       add tokenSet to res  
26       tokenSet ← ∅
```

---

---

**Listing 7.3** derived instance : SET(TOKENS) → INSTANCES

---

```
1 derived instance(tokens) =  
2 return res in  
3   choose t ∈ tokens do  
4     res ← instance(t)
```

---

---

**Listing 7.4** rule WorkflowTransition : FLOW\_NODES

---

```
1 rule WorkflowTransition(node) =  
2   parblock  
3     FlowNodeTransition(node)  
4     InstanceTransition(node)  
5   endparblock
```

---

## 7.2 Behaviors

In this section the different behaviors introduced in chapter 6 are defined formally. We start with the gate behavior introduced in section 6.2. Next, we continue with behaviors dealing with activation in section 7.2.2, formalizing the merge behaviors from section 6.3. Last, the split behaviors described in section 6.4 will be formalized in section 7.2.3.

### 7.2.1 Gate behavior

Derived function `allowedOutgoingSequenceFlows` : `FLOW_NODES`  $\rightarrow$  `SET(SEQUENCE_FLOWS)` shown in signature 7.5 is an abstract function used in `SplitBehavior` (see signature 7.9) to determine those outgoing sequence flows which will get a token. This abstract function has to be further refined in the concrete behaviors such as `ExclusiveSplitBehavior` (see listing 7.21), `InclusiveSplitBehavior` (see listing 7.22), and `ParallelSplitBehavior` (see listing 7.20).

```
abstract derived allowedOutgoingSequenceFlows : FLOW-  
_NODES  $\rightarrow$  SET(SEQUENCE_FLOWS) [7.5]
```

The derived function `canPass` shown in listing 7.5 defines the conditional flow possibilities for a BPMN diagram. The first parameter is the flow node the gate behavior is defined for, the second parameter holds the concrete outgoing sequence flow of that flow node, the third parameter identifies the instance of the flow node, and the forth parameter defines if the behavior should be "EXCLUSIVE", "INCLUSIVE", or "PARALLEL".

### 7.2.2 Activation behavior

The next two derived functions are regarding firing a flow node. The first, shown in signature 7.6, contains all tokens, which contributed to a fire of a flow node, and the later, shown in signature 7.7 defines a set of instances in which the given flow node fired.

```
abstract derived firingTokens : FLOW_NODES  $\rightarrow$  SET(-  
TOKENS) [7.6]
```

```
abstract derived firingInstances : FLOW_NODES  $\rightarrow$   
MULTISET(INSTANCES) [7.7]
```

---

**Listing 7.5** `derived canPass : FLOW_NODES × SEQUENCE_FLOWS × INSTANCES × GATE_BEHAVIOR → BOOLEAN`

---

```

1 derived canPass(flowNode, sequenceFlow, instance, behavior) =
2   return res in
3     let others ← outgoingSequenceFlows(flowNode) \ sequenceFlow in
4       if (behavior = "EXCLUSIVE" ∨ behavior = "INCLUSIVE")
5         ∧ gateConditionExpression(sequenceFlow) = "DEFAULT" then
6         res ← ∀ other ∈ others :
7           ¬ evaluate(gateConditionExpression(other), instance) holds
8
9       else if behavior = "EXCLUSIVE" then
10        res ← evaluate(gateConditionExpression(sequenceFlow), instance) = T
11        ∧ ∀ otherSequenceFlow ∈ { sf | sf ∈ others
12          ∧ gateConditionExpression(sf) ≠ "DEFAULT" } :
13          ¬ evaluate(gateConditionExpression(otherSequenceFlow),
14            instance) holds
14
15       else if behavior = "INCLUSIVE" then
16        res ← evaluate(gateConditionExpression(sequenceFlow), instance)
17
18       else if behavior = "PARALLEL" then
19        res ← T

```

---

The `InputBehavior` shown in listing 7.6 defines the input behavior for of a flow node. This rule returns all relevant instances in which the flow node should fire. The size of the multiset identifies the number of fires. The tokens, which contributed to any of the fires should be consumed (see listing 7.7).

This rule distinguishes between two basic cases. First is the case when only one incoming sequence flow is targeting the given flow node. Second case is when multiple sequence flows are targeting the given flow node. This may be seen as unnecessary and we explicitly distinguish between those case to relate to the BPMN [1] specification, which in some places makes this separation explicitly, e.g., in the case of `gatewayDirection` attribute in gateways. On a closer look it can be observed that all the merging behaviors would in case of only one incoming sequence flow behave in the same way as the `SoleInputBehavior` does. The only difference is that the `SoleInputBehavior` uses the `OneIncomingSequenceFlow` constraint restricting the number of incoming sequence flow targeting the given flow node to the cardinality one. This allows us to define flow nodes where only one incoming sequence flow would be a valid configuration by defining the `Merge Behavior` as `SoleInputBehavior`.

The rule `ActivationBehavior : FLOW_NODES → MULTISSET(-INSTANCES)` shown in listing 7.7 defines the abstract behavior for activation of a flow node. This rule returns all relevant instances in which the flow node fired. The size of the returned multiset identifies the number of fires. The tokens, which contributed to any of the fires are returned by the abstract derived function `firingTokens : FLOW_NODES → SET(TOKENS)` (see signature 7.6) and consumed here. The instances in which the given flow node fired are returned using the abstract derived function `firingInstances : FLOW_NODES → MULTISSET(-`

---

**Listing 7.6** rule InputBehavior : FLOW\_NODES → MULTISSET(-INSTANCES)

---

```

1 rule InputBehavior(flowNode) =
2   return res in
3     / Will chose merging behavior only for multiple incoming
4       \gls{Sequence Flow} /
5     if |incomingSequenceFlows(flowNode)| = 1 then
6       res ← SoleInputBehavior(flowNode)
7     else
8       res ← MergeBehavior(flowNode)

```

---

INSTANCES) (see signature 7.7). Those two abstract derived functions need to be further refined in subrules defining the concrete sole or merging behavior.

---

**Listing 7.7** rule ActivationBehavior : FLOW\_NODES → MULTISSET(-INSTANCES)

---

```

1 rule ActivationBehavior(flowNode) =
2   return res in
3     ∀ token ∈ firingTokens(flowNode) do
4       ConsumeToken(token)
5
6   res ← firingInstances(flowNode)

```

---

The sole input behavior shown in listing 7.8 refines the ActivationBehavior : FLOW\_NODES → MULTISSET(INSTANCES) rule (see listing 7.7) for the usage in flow nodes, which have, or are allowed to have, only one incoming sequence flow. In this case every token set in enablingTokenSets will contain exactly one token. Consequently, all tokens in all enablingTokenSets are firingTokens and every instance of every such firing token is a firing instance. Therefore, the function firingInstances holds instances of all tokens residing on the one incoming sequence flow.

---

**Listing 7.8** rule SoleInputBehavior : FLOW\_NODES → MULTISSET(-INSTANCES)

---

```

1 rule SoleInputBehavior(flowNode) =
2   assert OneIncomingSequenceFlow(flowNode)
3
4   return res in
5     let ft ← ⋃ enablingTokenSets(flowNode) in
6     res ← ActivationBehavior(flowNode) where
7       firingTokens(flowNode) = ft
8       firingInstances(flowNode) = [ i | ∃ t ∈ ft ∧ instance(t) = i ]

```

---

Will raise an error with the name of the constraint if the given flow node has more than one incoming sequence flows.

---

**Listing 7.9** constraint OneIncomingSequenceFlow : FLOW\_NODES
 

---

```

1 constraint OneIncomingSequenceFlow(flowNode) =
2   |incomingSequenceFlows(flowNode)| = 1

```

---

The MergeBehavior shown in signature 7.8 defines the abstract behavior pattern for merging multiple sequence flows of a flow node. This rule returns all relevant instances of a (sub-)process containing the given flow node in which the flow node fired.

abstract rule MergeBehavior : FLOW_NODES	[7.8]
--	-------

The rule ParallelMergeBehavior : FLOW\_NODES → MULTISSET(INSTANCES) shown in listing 7.10 refines the rule ActivationBehavior : FLOW\_NODES → MULTISSET(INSTANCES) (see signature B.8), for the usage in parallel gateways. This behavior defines firingTokenSets as enabling-TokenSets, which have tokens in all incoming sequence flows of the given flow node. The firingTokens are then all tokens in all such firingTokenSets and firingInstances are instances per firing token set (see listing 7.3).

---

**Listing 7.10** rule ParallelMergeBehavior : FLOW\_NODES → MULTISSET(INSTANCES)
 

---

```

1 rule ParallelMergeBehavior(flowNode) =
2   return res in
3     let firingTokenSets ← { ts | ts ∈ enablingTokenSets(flowNode)
4       ∧ ∀ sf ∈ incomingSequenceFlows(flowNode) : (
5         ∃ t ∈ ts with
6           t ∈ tokensInSequenceFlow(sf)) holds } in
7     res ← ActivationBehavior(flowNode) where
8       firingTokens(flowNode) = ⋃ firingTokenSets
9       firingInstances(flowNode) = [ i | ∃ ts ∈ firingTokenSets
10         ∧ instance(ts) = i ]

```

---

The rule ExclusiveMergeBehavior : FLOW\_NODES → MULTISSET(INSTANCES) shown in listing 7.11 refines the rule ActivationBehavior : FLOW\_NODES → MULTISSET(INSTANCES) (see signature B.8) for the usage in, e.g., exclusive gateways or activities. This behavior fires the given flow node if at least one token is present on any of the incoming sequence flows. No matter if only one or more tokens are present on the incoming sequence flows all will be consumed and for each of them the given flow node will fire. In other words it also means that this behavior will fire the given flow node for each token on any of the incoming sequence flows in any instance and may also fire multiple times in the same instance (see signature 7.7).

If a gateway is allowed to have only one incoming sequence flow containing a token the rule MutuallyExclusiveMergeBehavior shown in listing 7.12 should



---

**Listing 7.11** rule ExclusiveMergeBehavior : FLOW\_NODES → MULTISSET(INSTANCES)

---

```

1 rule ExclusiveMergeBehavior(flowNode) =
2   return res in
3     let ft ←  $\bigcup$  enablingTokenSets(flowNode) in
4     res ← ActivationBehavior(flowNode) where
5       firingTokens(flowNode) = ft
6       firingInstances(flowNode) = [ i |  $\exists t \in ft \wedge \text{instance}(t) = i$  ]

```

---

be used. Thus, more than one incoming sequence flow containing a token is considered as a violation of this rule and will raise an `MultipleTokensAtMutuallyExclusiveMerge` error. On the other hand multiple tokens in the same incoming sequence flow will not raise such an error. This is considered as correct behavior and will fire the given flow node for each token residing in such a sequence flow.

---

**Listing 7.12** rule MutuallyExclusiveMergeBehavior : FLOW\_NODES → MULTISSET(INSTANCES)

---

```

1 rule MutuallyExclusiveMergeBehavior(flowNode) =
2   assert MultipleTokensAtMutuallyExclusiveMerge(flowNode)
3
4   return res in
5     res ← ExclusiveMergeBehavior(flowNode)

```

---

The `MultipleTokensAtMutuallyExclusiveMerge` shown in listing 7.13 holds if only one incoming sequence flows has a token. More than one token on the same sequence flow is possible.

---

**Listing 7.13** constraint MultipleTokensAtMutuallyExclusiveMerge : FLOW\_NODES

---

```

1 constraint MultipleTokensAtMutuallyExclusiveMerge(flowNode) =
2    $\forall \text{tokenSet} \in \text{enablingTokenSets}(\text{flowNode}) : |\text{tokenSet}| \leq 1$  holds

```

---

The rule `InclusiveMergeBehavior : FLOW_NODES → MULTISSET(INSTANCES)` shown in listing 7.14 refines the rule `ActivationBehavior : FLOW_NODES → MULTISSET(INSTANCES)` and is used, e.g., in inclusive gateways. This behavior fires the given flow node if at least one token is present on any of the incoming sequence flows and no token in the process “may still arrive” [60]. This is defined as upstream token and the presence of such upstream token is checked using the `UpstreamTokens : FLOW_NODES × SET(SEQUENCE_FLOWS) × SET(TOKENS) → SET(TOKENS)` rule. If no such upstream token exists, this behavior will fire for every firing token set which meets the above condition.

The rule `ComplexMergeBehavior : FLOW_NODES → MULTISSET(INSTANCES)` shown in listing 7.15 takes, compared to other merge behaviors, an internal state of the complex gateway into account [1]. This state is also handled

---

**Listing 7.14** rule InclusiveMergeBehavior : FLOW\_NODES → MULTISSET(INSTANCES)

---

```

1 rule InclusiveMergeBehavior(flowNode) =
2   let firingTokenSets ← { ts |
3     UpstreamTokens(flowNode, incomingSequenceFlows(flowNode), ts) = 0 } in
4     ActivationBehavior(flowNode) where
5       firingTokens(flowNode) =  $\bigcup$  firingTokenSets
6       firingInstances(flowNode) = [ i |  $\exists$  ts  $\in$  firingTokenSets
7          $\wedge$  instance(ts) = i ]

```

---

only in this rule. It may be read for the purpose of conditions in outgoing sequence flows but not written. Firing a flow node using this ComplexMergeBehavior in waitingForStart conditioned only by the activationConditionExpression : COMPLEX\_GATEWAYS → EXPRESSIONS. In the “waiting for reset” state [1] the firing condition is similar to the InclusiveMergeBehavior : FLOW\_NODES → MULTISSET(INSTANCES) shown in listing 7.14. The difference is that the relevant sequence flows, where the flow node may wait for a token, which “may still arrive”, does not include the sequence flows which contributed to the activation in the first phase [1].

The BPMN 2.0 specification does not say anything about the quantity of tokens and the activationConditionExpression does not define any internal structure from which it would be possible to obtain the concrete tokens to be consumed. Therefore, we cannot use our enablingTokenSets in the way as other merge behaviors are using it and we define the ComplexMergeBehavior as follows (see listing 7.15: A set of possible firing instances ( $fi_P$ ) in which the flow node may fire is defined as a set of instances where there is an enabling token set for the given flow node in any of such instances. Then the firing instances in waitingForStart state ( $fi_S$ ) are all possible firing instances ( $fi_P$ ) in which the flow node is in waitingForStart state and the activationConditionExpression evaluates in such instances to true. The possible firing instances in “waiting for reset” state ( $fi_{RP}$ ) are all possible firing instances ( $fi_P$ ) in which the flow node is in not waitingForStart state. An important property of the firing instances in waitingForStart state ( $fi_S$ ) and the possible firing instances in “waiting for reset” state ( $fi_{RP}$ ) is that they are distinct sets due to the waitingForStart condition, i.e.,  $fi_S \cap fi_{RP} = \emptyset$ .

Next, we need to extract firing token sets from enablingTokenSets for each state. The first, firing token sets in waitingForStart state ( $fts_S$ ) are all enablingTokenSets which are in one of the firing instances in waitingForStart state ( $fi_S$ ). The second, firing token sets in “waiting for reset” state ( $fts_R$ ) which are in one of the possible firing instances in “waiting for reset” state ( $fi_{RP}$ ) and there is a token in such enabling token set which does not reside in one of the sequenceFlowsToIgnoreDuringReset. Since the firing instances in waitingForStart state ( $fi_S$ ) and the possible firing instances in “waiting for reset” state ( $fi_{RP}$ ) are two distinct sets and that all tokens in one enabling token set belong to the same instance (see listing 7.2), the firing token sets in waitingForStart state ( $fts_S$ ) and the firing token sets in “waiting for reset” state ( $fts_R$ ) are also

distinct sets, i.e.,  $fts_S \cap fts_R = \emptyset$ .

Now we can define the firing instances in “waiting for reset” state ( $fi_R$ ) as possible firing instances in “waiting for reset” state ( $fi_{RP}$ ) where there is no upstream token for the given flow node in relevant incoming sequence flows with relevant tokens residing on those relevant incoming sequence flows. Relevant incoming sequence flows are incoming sequence flows to the given flow node without `sequenceFlowsToIgnoreDuringReset`.

Next, all sequence flows which contributed to the activation in the `waitingForStart` state will be marked as `sequenceFlowsToIgnoreDuringReset` and the `waitingForStart` state will be set for all firing instances in `waitingForStart` state ( $fi_S$ ) to false and for all firing instances in “waiting for reset” state ( $fi_R$ ) to true.

Last, the `ActivationBehavior` (see listing 7.7) will be refined by defining the `firingTokens` as a union of all tokens from all firing token sets in `waitingForStart` state ( $fts_S$ ) and all tokens from firing token sets in “waiting for reset” state ( $fts_R$ ) which do not reside on `sequenceFlowsToIgnoreDuringReset`. The `firingInstances` of the `ActivationBehavior` are defined as a union of all firing instances in `waitingForStart` state ( $fi_S$ ) and all firing instances in “waiting for reset” state ( $fi_R$ ).

### 7.2.3 Output behavior

The rule `OutputBehavior` : `FLOW_NODES`  $\times$  `MULTISET(INSTANCES)` shown in listing 7.16 defines the abstract behavior for producing tokens on outgoing sequence flows of a flow node.

The rule `SoleOutputBehavior` : `FLOW_NODES`  $\times$  `INSTANCES` shown in listing 7.17 defines the behavior for producing a token for flow nodes with exactly one outgoing sequence flow. This is realized with the `OneOutgoingSequenceFlow` constraint.

Will raise an error with the name of the constraint if the given flow node has more than one outgoing sequence flows.

Abstract split behavior shown in signature 7.9 is used in other behaviors where it is refined to implement a concrete split behavior.

```
abstract rule SplitBehavior : FLOW_NODES  $\times$  INSTANCES [7.9]
```

The rule `Split` : `FLOW_NODES`  $\times$  `INSTANCES` shown in listing 7.19 defines the splitting of the control flow originating in the given flow node into parallel or alternative paths. This is done by producing a token on all `allowedOutgoingSequenceFlows` of the given flow node.

The `ParallelSplitBehavior` : `FLOW_NODES`  $\times$  `INSTANCES` shown in listing 7.20 refines the rule `SplitBehavior` : `FLOW_NODES`  $\times$  `INSTANCES`. This behavior is used as a default splitting behavior in BPMN 2.0 for any flow node

---

**Listing 7.15** rule ComplexMergeBehavior : FLOW\_NODES  $\rightarrow$  MULTISSET(-INSTANCES)

---

```

1 rule ComplexMergeBehavior(flowNode) =
2   return res in
3     let  $fi_P \leftarrow \{i \mid \exists ts \in \text{enablingTokenSets}(\text{flowNode}) \wedge \text{instance}(ts) = i\}$  in
4     let  $fi_S \leftarrow \{i \mid \exists i \in fi_P : \text{waitingForStart}(\text{flowNode}, i) = \top$ 
5        $\wedge \text{evaluate}(\text{activationConditionExpression}(\text{flowNode}), i) = \top \text{ holds}$ 
6        $\}$ ,
7      $fi_{RP} \leftarrow \{i \mid \exists i \in fi_P : \text{waitingForStart}(\text{flowNode}, i) = \perp\}$  in
8     let  $fts_S \leftarrow \{ts \mid$ 
9        $ts \in \text{enablingTokenSets}(\text{flowNode}) \wedge \text{instance}(ts) \in fi_S\}$ ,
10     $fts_R \leftarrow \{ts \mid ts \in \text{enablingTokenSets}(\text{flowNode}) \wedge \text{instance}(ts) \in fi_{RP}$ 
11       $\wedge \exists t \in ts : \text{sequenceFlow}(t) \notin \text{sequenceFlowsToIgnoreDuringReset}(\text{flowNode}, \text{instance}(ts)) \text{ holds}\}$  in
12    let  $fi_R \leftarrow \{i \mid i \in fi_{RP}$ 
13       $\wedge \text{UpstreamTokens}(\text{flowNode},$ 
14         $\text{incomingSequenceFlows}(\text{flowNode}) \setminus$ 
15         $\text{sequenceFlowsToIgnoreDuringReset}(\text{flowNode}, i),$ 
16         $\{t \mid t \in \text{UNION } fts_R$ 
17           $\wedge t \notin \text{sequenceFlowsToIgnoreDuringReset}(\text{flowNode}, i)\} =$ 
18           $\emptyset\}$  in
19
20     $\forall i \in fi_S \text{ do}$ 
21       $\text{sequenceFlowsToIgnoreDuringReset}(\text{flowNode}, i) \leftarrow \{sf \mid$ 
22         $sf \in \text{incomingSequenceFlows}(\text{flowNode})$ 
23         $\wedge \exists ts \in fts_S$ 
24         $\wedge \exists t \in ts : \text{sequenceFlow}(t) = sf \text{ holds}\}$ 
25       $\text{waitingForStart}(\text{flowNode}, i) \leftarrow \perp$ 
26
27     $\forall i \in fi_R \text{ do}$   $\text{waitingForStart}(\text{flowNode}, i) \leftarrow \top$ 
28
29    res  $\leftarrow \text{ActivationBehavior}(\text{flowNode})$  where
30       $\text{firingTokens}(\text{flowNode}) = (\text{UNION } fts_S) \cup \{t \mid t \in \text{UNION } fts_R$ 
31         $\wedge t \notin \text{sequenceFlowsToIgnoreDuringReset}(\text{flowNode}, i)\}$ 
32       $\text{firingInstances}(\text{flowNode}) = \bigcup (fi_S \cup fi_R)$ 

```

---



---

**Listing 7.16** rule OutputBehavior : FLOW\_NODES  $\times$  MULTISSET(-INSTANCES)

---

```

1 rule OutputBehavior(flowNode, instances) =
2    $\forall \text{instance} \in \text{instances} \text{ do}$ 
3     if  $|\text{outgoingSequenceFlows}(\text{flowNode})| = 1$  then
4        $\text{SoleOutputBehavior}(\text{flowNode}, \text{instance})$ 
5     else
6        $\text{SplitBehavior}(\text{flowNode}, \text{instance})$ 

```

---

---

**Listing 7.17** rule `SoleOutputBehavior` : `FLOW_NODES`  $\times$  `INSTANCES`

---

```
1 rule SoleOutputBehavior(flowNode, instance) =  
2   assert OneOutgoingSequenceFlow(flowNode)  
3  
4   choose sequenceFlow  $\in$  outgoingSequenceFlows(flowNode) do  
5     Split(flowNode, instance) where  
6       allowedOutgoingSequenceFlows(flowNode) = { sequenceFlow }
```

---

---

**Listing 7.18** constraint `OneOutgoingSequenceFlow` : `FLOW_NODES`

---

```
1 constraint OneOutgoingSequenceFlow(flowNode) =  
2   |outgoingSequenceFlows(flowNode)| = 1
```

---

---

**Listing 7.19** rule `Split` : `FLOW_NODES`  $\times$  `INSTANCES`

---

```
1 rule Split(flowNode, instance) =  
2    $\forall$  sequenceFlow  $\in$  allowedOutgoingSequenceFlows(flowNode) do  
3     ProduceToken(sequenceFlow, instance)
```

---

except exclusive, inclusive, complex and event-based gateways. This behavior defines all outgoing sequence flows as allowed and will produce a token on all of them (see: listing 7.5).

---

**Listing 7.20** rule `ParallelSplitBehavior` : `FLOW_NODES`  $\times$  `INSTANCES`

---

```
1 rule ParallelSplitBehavior(flowNode, instance) =  
2   Split(flowNode, instance) where  
3     allowedOutgoingSequenceFlows(flowNode) =  
4     { sequenceFlow | sequenceFlow  $\in$  outgoingSequenceFlows(flowNode)  
5        $\wedge$  canPass(flowNode, sequenceFlow, instance, "PARALLEL") }
```

---

The `ExclusiveSplitBehavior` : `FLOW_NODES`  $\times$  `INSTANCES` shown in listing 7.21 refines the `SplitBehavior` : `FLOW_NODES`  $\times$  `INSTANCES`, which is used in exclusive gateways. This behavior allows to produce a token on exactly one outgoing sequence flow as defined in `canPass`.

---

**Listing 7.21** rule `ExclusiveSplitBehavior` : `FLOW_NODES`  $\times$  `INSTANCES`

---

```
1 rule ExclusiveSplitBehavior(flowNode, instance) =  
2   Split(flowNode, instance) where  
3     allowedOutgoingSequenceFlows(flowNode) =  
4     { sequenceFlow | sequenceFlow  $\in$  outgoingSequenceFlows(flowNode)  
5        $\wedge$  canPass(flowNode, sequenceFlow, instance, "EXCLUSIVE") }
```

---

The `InclusiveSplitBehavior` : `FLOW_NODES`  $\times$  `INSTANCES` shown in listing 7.22 refines the rule `SplitBehavior` : `FLOW_NODES`  $\times$  `INSTANCES`,

which is used in inclusive and complex gateways. It allows to produce a token on any subset of the outgoing sequence flows (see: listing 7.5).

---

**Listing 7.22** rule InclusiveSplitBehavior : FLOW\_NODES  $\times$  INSTANCES

---

```

1 rule InclusiveSplitBehavior(flowNode, instance) =
2   Split(flowNode, instance) where
3     allowedOutgoingSequenceFlows(flowNode) =
4       { sequenceFlow | sequenceFlow  $\in$  outgoingSequenceFlows(flowNode)
5          $\wedge$  canPass(flowNode, sequenceFlow, instance, "INCLUSIVE") }

```

---

### 7.3 Flow nodes

In this section we will define the flow node specific functions and rules.

The first two derived functions handle the incoming or outgoing sequence flows into or out of the given flow node. The first is shown in listing 7.23 and uses the static function `static targetFlowNode : SEQUENCE_FLOWS  $\rightarrow$  FLOW_NODES` (see signature 7.2). The later is shown in listing 7.24 and uses the static function `static sourceFlowNode : SEQUENCE_FLOWS  $\rightarrow$  FLOW_NODES` (see signature 7.1).

---

**Listing 7.23** derived incomingSequenceFlows : FLOW\_NODES  $\rightarrow$  SET(-SEQUENCE\_FLOWS)

---

```

1 derived incomingSequenceFlows(flowNode) =
2   { x | x  $\in$  SEQUENCE_FLOWS  $\wedge$  targetFlowNode(x) = flowNode }

```

---



---

**Listing 7.24** derived outgoingSequenceFlows : FLOW\_NODES  $\rightarrow$  SET(-SEQUENCE\_FLOWS)

---

```

1 derived outgoingSequenceFlows(flowNode) =
2   { x | x  $\in$  SEQUENCE_FLOWS  $\wedge$  sourceFlowNode(x) = flowNode }

```

---

Some flow nodes can be encapsulated in other flow nodes. This is, e.g., in the case of sub-processes. For this purpose we define the static function `static parentFlowNode : FLOW_NODES  $\rightarrow$  ACTIVITIES` shown in signature 7.10.

`static parentFlowNode : FLOW_NODES  $\rightarrow$  ACTIVITIES` [7.10]

The rule `FlowNodeTransition : FLOW_NODES` checks required conditions for a flow node regarding events, control flow, data, and resources, and performs the corresponding operations. This rule represents the most abstract transition rule for

flow nodes and is further refined on lower levels for activities, events, and gateways, and further for specific types of these flow nodes.

---

**Listing 7.25** rule FlowNodeTransition : FLOW\_NODES

---

```

1 rule FlowNodeTransition(flowNode) =
2   if activationCondition(flowNode)
3      $\wedge$  dataCondition(flowNode)
4      $\wedge$  resourceCondition(flowNode)
5   then parblock
6     FlowNodeOperation(flowNode)
7     DataOperation(flowNode)
8     ResourceOperation(flowNode)
9   endparblock

```

---

Inside the FlowNodeTransition rule the following three abstract condition functions are defined: the abstract derived activationCondition : FLOW\_NODES  $\rightarrow$  BOOLEAN (see signature 7.11), the abstract derived dataCondition : FLOW\_NODES  $\rightarrow$  BOOLEAN (see signature 7.12), and the abstract derived resourceCondition : FLOW\_NODES  $\rightarrow$  BOOLEAN (see signature 7.13). This is a significant difference between this work and the work in [16, 17] where we use a common activationCondition where in the mentioned work the authors differentiate between controlCondition and eventCondition and further between ControlOperation and EventOperation while we use a common activationCondition for the first and a common FlowNodeOperation for the latter. We differentiate between control and event specific condition and operation later regarding events as discussed in section 6.7. The first, activationCondition, determines if activation conditions, such as tokens on the incoming sequence flows of the given flow node (only if such a flow node has incoming sequence flows) or occurring events (only in case of event nodes), are met. The second, dataCondition, determines if data constraints are met before activating the given flow node. It is currently not specified for any transition rule in the current ground model and has to be refined as soon as the data aspect is included. The last, resourceCondition, determines if resource constraints are met before activating the given flow node. Currently it is also kept abstract for all transitions because the BPMN standard does not sufficiently specify resources. We keep the data and resource related conditions and operations in the ground model to present the relation to work in [16, 17] we base our ground model on and also to keep in mind that those aspects should be target of further refinements.

<pre> abstract derived activationCondition : FLOW_NODES <math>\rightarrow</math> BOOLEAN </pre> <div style="text-align: right; padding-right: 20px;">[7.11]</div>
---

```
abstract derived dataCondition : FLOW_NODES →  
BOOLEAN [7.12]
```

```
abstract derived resourceCondition : FLOW_NODES →  
BOOLEAN [7.13]
```

For each of the above condition function a corresponding operation rule is defined. The first abstract rule `FlowNodeOperation` shown in signature 7.14 is responsible for performing the work that has to be done in a flow node, e.g., creating instances of tasks and sub-processes, producing a token or throwing an event. The second abstract rule `DataOperation` shown in signature 7.15 is responsible for data. It is currently not specified for any transition in the current ground model, and has to be refined as soon as the data aspect is included. The last abstract rule `ResourceOperation` shown in signature 7.16 is responsible for resources. Currently it is kept abstract for all transitions because the BPMN standard does not sufficiently specify resources. All of these rules have to be refined for all transitions of non-abstract meta-model classes.

```
abstract rule FlowNodeOperation : FLOW_NODES [7.14]
```

```
abstract rule DataOperation : FLOW_NODES [7.15]
```

```
abstract rule ResourceOperation : FLOW_NODES [7.16]
```

This is defining the abstract *FlowNode* from the meta-model. Next we will into the different flow node type specific functions and rules.

## 7.4 Activities

In this section we show the activity specific functions and rules. They are based on the attributes of the *Activity* class and on the related part of the meta-model in [1].

The monitored function `completed` shown in signature 7.17 indicates if the given activity in a given instance has been completed. This location must be set, e.g., for tasks when their work is finished.



$$\text{shared completed} : \text{ACTIVITIES} \times \text{INSTANCES} \rightarrow \text{BOOLEAN} \quad [7.17]$$

The monitored function `interrupted` shown in signature 7.18 indicates if the given activity in a given instance has been interrupted.

$$\text{monitored interrupted} : \text{ACTIVITIES} \times \text{INSTANCES} \rightarrow \text{BOOLEAN} \quad [7.18]$$

The controlled function `lifeCycleState` shown in signature 7.19 gives the instance life-cycle state of an activity [1, tab. 10.4]. Each time a token arrives on an incoming sequence flow of an activity a new instance of the target activity will be created with the initial life-cycle state “Ready”. The `lifeCycleState` function may take one of the following values in case no errors or interruptions occur. During the run of one instance of an activity the life-cycle starts with the initial “Ready” state. Next, the activity will transit to the “Active” state when the data conditions are met. After the activity’s work was completed it will transit to the “Completing” state. Then, if the activity was not interrupted, it will transit to the “Completed” state after all completing requirements are met and assignments are completed. Finally, if no compensation occurred the activity will finish in the “Closed” state. In case of an error, interruption or compensation during the activity instance life-cycle run the activity may be found in one of the following life-cycle states: “Failing”, “Terminating”, “Compensating”, “Failed”, “Terminated”, “Compensated” and “Withdrawn”. For detailed information about the activity life-cycle states and the transitions between them see [1, sec. 13.2.2].

$$\text{controlled lifeCycleState} : \text{ACTIVITIES} \times \text{INSTANCES} \rightarrow \text{LIFE\_CYCLE\_STATES} \quad [7.19]$$

The controlled function `instantiatingFlowNode` shown in signature 7.20 holds a flow node for which the given instance has been created. This is used, e.g., for sub-processes.

$$\text{controlled instantiatingFlowNode} : \text{INSTANCES} \rightarrow \text{FLOW\_NODES} \quad [7.20]$$

The controlled function `parentInstance` shown in signature 7.21 holds the parent instance of the given instance. This may be `undef` for top-level process instances.

$$\text{controlled parentInstance} : \text{INSTANCES} \rightarrow \text{INSTANCES} \quad [7.21]$$

The rule `InstanceTransition` : `ACTIVITIES` shown in listing 7.26 handles the life-cycle states of all relevant instances of the given activity using the `InstanceOperation` shown in signature 7.22.

---

**Listing 7.26** rule `InstanceTransition` : `ACTIVITIES`

---

```

1 rule InstanceTransition(activity) =
2    $\forall$  instance  $\in$  relevantInstances(activity) do
3     InstanceOperation(instance, activity)

```

---

The abstract rule `InstanceOperation` shown in signature 7.22 is responsible for instance related work. This rule is refined only for activities since other flow nodes cannot be instantiated.

$$\text{abstract rule InstanceOperation} : \text{INSTANCES} \times \text{ACTIVITIES} \quad [7.22]$$

The rule `ActivityTransition` shown in listing 7.27 refines the `control-Operation` for activities if the start quantity for enabling tokens is satisfied. Then first, an instance of the activity is created by consuming the enabling tokens. Afterwards, the instance of the activity is executed by the rule `GetActive` that calls the rule `StartOperation` which differentiates various types of activities. See [44] for detailed information. This will not be further refined in this thesis.

The rule `CreateInstance` shown in listing 7.28 creates a new instance of an activity. Additionally, the new instance is added to the function `activeInstances` : `ACTIVITIES`. If a new instance of a top-level process is needed, then the `Dispatch` rule should be used. No token is created within this rule, because it will be created when the flow node is left.

We do not further refine the transition rules for any of the subclasses of the *Activity* class in the meta-model [1, fig. 10.6]. The goal is to show decomposition and refinement method enhancements rather than a complete BPMN ground model. We will continue with the event related functions and rules in the next section.

---

**Listing 7.27** rule ActivityTransition : ACTIVITIES

---

```
1 rule ActivityTransition(flowNode) =  
2   if flowNode ∈ ACTIVITIES then  
3     let firingInstances ← ExclusiveMergeBehavior(flowNode) in  
4       FlowNodeTransition(flowNode) where  
5         activationCondition(flowNode) = firingInstances ≠ ∅  
6         FlowNodeOperation(flowNode) =  
7           ∀ instance ∈ firingInstances do  
8             CreateInstance(flowNode, instance)  
9  
10      InstanceTransition(flowNode) where  
11        InstanceOperation(instance, flowNode) = parblock  
12          if lifeCycleState(instance, flowNode) = "Ready" then  
13            GetActive(instance, flowNode)  
14  
15          if lifeCycleState(instance, flowNode) ∈ {"Completed",  
16            "Compensated", "Failed", "Terminated", "Withdrawn"}  
17          then parblock  
18            ExitActivity(instance, flowNode)  
19          endparblock  
20        endparblock
```

---

---

**Listing 7.28** rule CreateInstance : ACTIVITIES × INSTANCES

---

```
1 rule CreateInstance(activity, parent) =  
2   return instance in  
3     instance ← new INSTANCES  
4     add instance to activeInstances(activity)  
5     instantiatingFlowNode(instance) ← activity  
6     parentInstance(instance) ← parent  
7     lifeCycleState(activity, instance) ← "Ready"
```

---

## 7.5 Events

In this section we show the event specific functions and rules. They are based on the attributes of the *Event* class and on the related part of the meta-model in [1].

The static function `triggers` shown in signature 7.23 replaces the related attributes `eventDefinitions` and `eventDefinitionRefs` from the original meta-model [1, tab. 10.82]. The name change is because of the name clarification done in section 5.2.3. From the execution point of view this function is the union of the two replaced attributes.

<code>static triggers : EVENTS → SET(TRIGGERS)</code>	[7.23]
---	--------

The static function `parallelMultiple` shown in signature 7.24 determines whether all defined triggers in an event have to occur in order the event fires or just one. It is only relevant when the catch event has more than one triggers defined [1, tab. 10.].

<code>static parallelMultiple : CATCH_EVENTS → BOOLEAN</code>	[7.24]
---	--------

The static function `attachedTo` shown in signature 7.25 denotes the activity that the given boundary event is attached to [1, tab. 10.91].

<code>static attachedTo : BOUNDARY_EVENTS → ACTIVITIES</code>	[7.25]
---	--------

The static function `conditionExpression` shown in signature 7.26 holds the condition expression for conditional triggers and timer triggers.

<code>static conditionExpression : CONDITIONAL_TRIGGERS → EXPRESSIONS</code>	[7.26]
--	--------

The controlled function `eventOccured` shown in signature 7.27 holds the information about event occurrence for a given flow node in a concrete instance.

<code>controlled eventOccured : EVENTS × INSTANCES → BOOLEAN</code>	[7.27]
---	--------

The derived function `triggerName : EVENTS → STRINGS` shown in listing 7.29 checks `triggers` (see signature 7.23) of the given event and returns "None"

if no trigger is defined for such an event. If more triggers are defined, it returns either "Multiple" or "ParallelMultiple" in case that the `parallelMultiple` (see signature 7.24) is set to true. Otherwise the appropriate trigger name ("Message", "Timer", "Error", etc.) is returned [1].

---

**Listing 7.29** `derived triggerName : EVENTS → STRINGS`

---

```

1 derived triggerName(event) =
2   return res in
3     if triggers(event) = 0 then
4       res ← "None"
5     else if |triggers(event)| = 1 then
6       choose trigger ∈ triggers(event) do
7         res ← triggerName(trigger)
8
9     else if parallelMultiple(event) then
10      res ← "ParallelMultiple"
11    else
12      res ← "Multiple"

```

---

The derived function `triggerName : TRIGGERS → STRINGS` shown in listing 7.30 returns the appropriate name ("Message", "Timer", "Error", etc.) for the given trigger [1].

---

**Listing 7.30** `derived triggerName : TRIGGERS → STRINGS`

---

```

1 derived triggerName(trigger) =
2   return res in
3     if trigger ∈ MESSAGE_TRIGGERS then
4       res ← "Message"
5     else if trigger ∈ TIMER_TRIGGERS then
6       res ← "Timer"
7     else if trigger ∈ ERROR_TRIGGERS then
8       res ← "Error"
9     else if trigger ∈ ESCALATION_TRIGGERS then
10      res ← "Escalation"
11     else if trigger ∈ CANCEL_TRIGGERS then
12      res ← "Cancel"
13     else if trigger ∈ COMPENSATION_TRIGGERS then
14      res ← "Compensation"
15     else if trigger ∈ CONDITIONAL_TRIGGERS then
16      res ← "Conditional"
17     else if trigger ∈ LINK_TRIGGERS then
18      res ← "Link"
19     else if trigger ∈ SIGNAL_TRIGGERS then
20      res ← "Signal"
21     else if trigger ∈ TERMINATE_TRIGGERS then
22      res ← "Terminate"

```

---

The `EventTransition : EVENTS` rule shown in listing 7.31 refines the `WorkflowTransition : FLOW_NODES` rule shown in listing 7.4 and defines the `activationCondition` using the `SelectFiringInstances` rule shown in list-

ing 7.32 and the `FlowNodeOperation` using an abstract `ControlOperation` for operations, i.e., production of tokens on outgoing sequence flows and an abstract `EventOperation`, i.e., throwing an event.

---

**Listing 7.31** rule `EventTransition` : `EVENTS`

---

```

1 rule EventTransition(event) =
2   let firingInstances ← SelectFiringInstances(event) in
3     WorkflowTransition(event) where
4       activationCondition(event) = firingInstances ≠ 0
5       FlowNodeOperation(event) =
6         ControlOperation(event, firingInstances)
7         EventOperation(event, firingInstances)

```

---

The rule `SelectFiringInstances` shown in listing 7.32 separates the control condition from event condition for events. This separation is defined from here since events are the only flow nodes which actually give the event condition use and therefore it is not necessary to model this separation before.

The `ControlActivationBehavior` is responsible for the control condition and consuming tokens. Since tokens, which satisfy the control condition are consumed immediately the instances in which a token configuration satisfied the control condition will be cached using the `waitingControlInstances` function. Similarly occurred events will be immediately consumed and the instances in which they occurred will be cached using the `waitingEventInstances` function.

There are three possibilities of what can happen. First, the concrete event type waits for both the control and the event condition to be satisfied, i.e., the event type defines both the `ControlActivationBehavior` and the `EventActivationBehavior`. In this case the selected instances, which can actually fire the given event, are a union of the two cached instance sets. In other words the event can fire only in instances in which both, the control condition and the event condition, were satisfied.

The second and third possibility is that either only the control or the event condition needs to be satisfied. I.e., only one of the two behaviors (`ControlActivationBehavior` or `EventActivationBehavior`) is defined and the other yields `undef`. In this case also the related cache function yields `undef` and is ignored and only instances in the other cache function are relevant.

The rule `CatchEventTransition` : `CATCH_EVENTS` shown in listing 7.33 refines the rule `EventTransition` : `EVENTS` (see listing 7.31) and defines only the event related parts of a flow node. A catch event defines `EventActivationBehavior` by checking if the event defined by a trigger related to the given flow node occurred. This is realized with the `eventOccurred` : `FLOW_NODES × INSTANCES → BOOLEAN` function (see signature 7.27). A catch event does not perform any `EventOperation` since no event is further thrown in this type of event node.

The rule `ThrowEventTransition` : `THROW_EVENTS` shown in listing 7.34 refines the rule `EventTransition` : `EVENTS` (see listing 7.31) and defines only the event related parts of a flow node. A throw event is never activated by an event

---

**Listing 7.32** rule SelectFiringInstances : EVENTS  $\rightarrow$  MULTISSET(-INSTANCES)

---

```

1 rule SelectFiringInstances(event) =
2   local allWaitingControlInstances  $\leftarrow$  waitingControlInstances(event)
3      $\cup$  ControlActivationBehavior(event),
4     allWaitingEventInstances  $\leftarrow$  waitingEventInstances(event)
5      $\cup$  EventActivationBehavior(event) in
6   return res in
7     if waitingControlInstances(event)  $\neq$  undef
8        $\wedge$  waitingEventInstances(event)  $\neq$  undef then
9       res  $\leftarrow$  allWaitingControlInstances
10         $\cap$  allWaitingEventInstances
11   else if waitingControlInstances(event)  $\neq$  undef
12      $\wedge$  waitingEventInstances(event) = undef then
13     res  $\leftarrow$  allWaitingControlInstances
14   else if waitingControlInstances(event) = undef
15      $\wedge$  waitingEventInstances(event)  $\neq$  undef then
16     res  $\leftarrow$  allWaitingEventInstances
17   else
18     res  $\leftarrow$   $\emptyset$ 
19
20   waitingControlInstances(event)  $\leftarrow$  allWaitingControlInstances  $\setminus$  res
21   waitingEventInstances(event)  $\leftarrow$  allWaitingEventInstances  $\setminus$  res

```

---



---

**Listing 7.33** rule CatchEventTransition : CATCH\_EVENTS

---

```

1 rule CatchEventTransition(event) =
2   EventTransition(event) where
3     EventActivationBehavior(event) =
4     return res in
5       let instances  $\leftarrow$  { i | i  $\in$  INSTANCES  $\wedge$  eventOccured(event, i) } in
6        $\forall$  instance  $\in$  instances do
7         eventOccured(event, instance)  $\leftarrow$   $\perp$ 
8       res  $\leftarrow$  instances
9
10   EventOperation(event, instances) = skip

```

---

and therefore the `EventActivationBehavior` yields an `undef`. On the other hand, a throw event does perform `EventOperation` using `Throw` rule (see signature 7.28) in all instances it was fired.

---

**Listing 7.34** rule `ThrowEventTransition` : `THROW_EVENTS`

---

```

1 rule ThrowEventTransition(event) =
2   EventTransition(event) where
3     EventActivationBehavior(event) = undef
4     EventOperation(event, instances) =
5       ∀ instance ∈ instances do
6         ∀ trigger ∈ triggers(event) do
7           Throw(trigger, instance)

```

---

The abstract rule `Throw` : `TRIGGERS` × `INSTANCES` shown in signature 7.28 is meant to be fired for every trigger of throw event in a given instance.

abstract rule <code>Throw</code> : <code>TRIGGERS</code> × <code>INSTANCES</code>	[7.28]
---	--------

The `StartEventTransition` : `START_EVENTS` show in listing 7.35 refines the rule `CatchEventTransition` : `CATCH_EVENTS` (see listing 7.33), which defines the event related condition and operation. Whether the given flow node (event in this case) fires depends in this case only on the event condition and therefore the `ControlActivationBehavior` yields `undef`.

If a start event is a source of multiple outgoing sequence flows [1, sec. 10.4.2] and such sequence flows then form parallel paths, we reuse the `ParallelSplitBehavior` : `FLOW_NODES` × `INSTANCE` (see listing 7.20) to realize this. Additionally a start event creates a new instance of the encompassing activity.

---

**Listing 7.35** rule `StartEventTransition` : `START_EVENTS`

---

```

1 rule StartEventTransition(event) =
2   CatchEventTransition(event) where
3     ControlActivationBehavior(event) = undef
4     ControlOperation(event, instances) =
5       ∀ parent ∈ instances do
6         let instance ← CreateInstance(parentFlowNode(event), parent) in
7           ParallelSplitBehavior(event, instance)

```

---

The rule `EndEventTransition` : `END_EVENTS` shown in listing 7.36 refines the rule `ThrowEventTransition` : `THROW_EVENTS` (see listing 7.34), which defines the event related condition and operation. Whether the given event fires depends in this case also on the control condition which is defined as `ExclusiveMergeBehavior` (see listing 7.11). A end event has no outgoing sequence flows and therefore the `ControlOperation` does not perform anything.

The rule `IntermediateEventTransition` : `INTERMEDIATE_EVENTS` shown in listing 7.37 refines the rule `EventTransition` : `EVENTS` (see list-



---

**Listing 7.36** rule EndEventTransition : END\_EVENTS

---

```
1 rule EndEventTransition(event) =  
2   ThrowEventTransition(event) where  
3     ControlActivationBehavior(event) = ExclusiveMergeBehavior(event)  
4     ControlOperation(event, instances) = skip
```

---

ing 7.31) and defines the control condition by defining the ControlActivationBehavior as ExclusiveMergeBehavior (see listing 7.11 and the ControlOperation using the ParallelSplitBehavior (see listing 7.20 for every firing instance.

The event related condition and operation will be further defined in subrules IntermediateCatchEventTransition (see listing 7.38) and IntermediateThrowEventTransition (see listing 7.39).

---

**Listing 7.37** rule IntermediateEventTransition : INTERMEDIATE\_EVENTS

---

```
1 rule IntermediateEventTransition(event) =  
2   EventTransition(event) where  
3     ControlActivationBehavior(event) = ExclusiveMergeBehavior(event)  
4     ControlOperation(event, instances) =  
5       ∀ instance ∈ instances do  
6         ParallelSplitBehavior(event, instance)
```

---

The rule IntermediateCatchEventTransition : INTERMEDIATE\_CATCH\_EVENTS shown in listing 7.38 is defined by the rule CatchEventTransition : CATCH\_EVENTS (see listing 7.33), which specifies the event condition and operation, and the rule IntermediateEventTransition : - INTERMEDIATE\_EVENTS (see listing 7.37), which specifies the control condition and operation.

---

**Listing 7.38** rule IntermediateCatchEventTransition : - INTERMEDIATE\_CATCH\_EVENTS

---

```
1 rule IntermediateCatchEventTransition(event) =  
2   CatchEventTransition(event)  
3   IntermediateEventTransition(event)
```

---

The rule IntermediateThrowEventTransition : INTERMEDIATE\_THROW\_EVENTS shown in listing 7.39 is defined by the rule ThrowEventTransition : THROW\_EVENTS (see listing 7.34), which specifies the event condition and operation, and the rule IntermediateEventTransition : - INTERMEDIATE\_EVENTS (see listing 7.37), which specifies the control condition and operation. the control condition and operation.

The rule BoundaryEventTransition : BOUNDARY\_EVENTS shown in listing 7.40 refines the rule CatchEventTransition : CATCH\_EVENTS (see list-

---

**Listing 7.39** rule IntermediateThrowEventTransition : -  
INTERMEDIATE\_THROW\_EVENTS

---

```

1 rule IntermediateThrowEventTransition(event) =
2   ThrowEventTransition(event)
3   IntermediateEventTransition(event)

```

---

ing 7.33) and defines the missing control condition and operation. The boundary event is not a target of any incoming sequence flows and therefore it does not wait for any tokens. This is modeled by the `ControlActivationBehavior` yielding `undef`. The boundary event can be source of one or more sequence flows in which case it will produce tokens on each of them in case the event fires. This is modeled using the `ParallelSplitBehavior` (see listing 7.20).

---

**Listing 7.40** rule BoundaryEventTransition : BOUNDARY\_EVENTS

---

```

1 rule BoundaryEventTransition(event) =
2   CatchEventTransition(event) where
3     ControlActivationBehavior(event) = undef
4     ControlOperation(event, instances) =
5       ∀ instance ∈ instances do
6         ParallelSplitBehavior(event, instance)

```

---

## 7.6 Gateways

In this section we show the gateway specific functions and rules. They are based on the attributes of the *Gateway* class and on the related part of the meta-model in [1].

The static function `gateConditionExpression` shown in signature 7.29 represents the gating condition for sequence flows. A token will only pass the gate on its outgoing sequence flow if this condition holds. In the original BPMN 2.0 specification this was implemented in sequence flows as `conditionExpression` → EXPRESSIONS [68, tab. 8.51], but since the gating refinement this parameter was renamed for better indication of its meaning and the placement of the actual condition.

```

static gateConditionExpression : SEQUENCE_FLOWS →
EXPRESSIONS

```

[7.29]

The static function `activationConditionExpression` shown in signature 7.30 determines which combination of incoming tokens will be synchronized for activation of the gateway [1, tab. 10.125].

```
static activationConditionExpression : COMPLEX_GATEWAYS
→ EXPRESSIONS [7.30]
```

The internal state of a complex gateway is represented by the controlled function `waitingForStart` shown in signature 7.31 [1, tab. 10.126].

```
controlled waitingForStart : COMPLEX_GATEWAYS × INSTANCES
→ BOOLEAN [7.31]
```

The controlled function `sequenceFlowsToIgnoreDuringReset` shown in signature 7.32 holds a set of sequence flows, which activated the complex gateway and should be ignored during the reset inclusive behavior.

```
controlled sequenceFlowsToIgnoreDuringReset : COMPLEX-
_GATEWAYS × INSTANCES → SET(SEQUENCE_FLOWS) [7.32]
```

The rule `GatewayTransition` shown in signature B.43 represents the common refinement of `WorkflowTransition` (see listing 7.4) for the gateway flow node.

---

**Listing 7.41** rule `GatewayTransition` : GATEWAYS

---

```
1 rule GatewayTransition(gateway) =
2   FlowNodeTransition(gateway)
```

---

The rule `DataBasedGatewayTransition` : DATA\_BASED\_GATEWAYS shown in listing 7.42 refines the rule `FlowNodeTransition` : FLOW\_NODES (see listing 7.25) and defines the `controlCondition` : FLOW\_NODES, `eventCondition` : FLOW\_NODES, `ControlOperation` : FLOW\_NODES, and `EventOperation` : FLOW\_NODES. The event related refinements are not relevant for a gateway and therefore the derived function `eventCondition` always returns true, while the rule `EventOperation` does not perform anything.

The derived function `controlCondition` : FLOW\_NODES defines the condition the gateway needs to hold in order to fire. This is realized using the `MergeBehavior` : FLOW\_NODES (see signature 7.8). The concrete gateway type will then use one of the defined `ParallelMergeBehavior` : - FLOW\_NODES, `ExclusiveMergeBehavior` : FLOW\_NODES, `InclusiveMergeBehavior` : FLOW\_NODES, or `ComplexMergeBehavior` : FLOW\_NODES.

The output part is realized by refining the rule `ControlOperation` : FLOW\_NODES. There one of the relevant tokens, which contributed to firing the gateway, will be chosen to determine the instance of the running process. Any of the tokens in the `enablingTokens` set may be chosen, since by design all of them have to

be from the same instance. This instance will then be passed to the rule `SplitBehavior : FLOW_NODES × INSTANCES` to perform the token production on the outgoing sequence flows of the gateway. The concrete gateway type will then use one of the defined `ExclusiveSplitBehavior : FLOW_NODES × INSTANCES` or `InclusiveSplitBehavior : FLOW_NODES × INSTANCES`.

---

**Listing 7.42** rule `DataBasedGatewayTransition : DATA_BASED_GATEWAYS`

---

```

1 rule DataBasedGatewayTransition(gateway) =
2   let firingInstances ← GatewayBehavior(gateway) in
3     GatewayTransition(gateway) where
4       activationCondition(gateway) = firingInstances ≠ 0
5       FlowNodeOperation(gateway) = GatewayOperation(gateway, firingInstances)

```

---

The rule `ParallelGatewayTransition : PARALLEL_GATEWAYS` shown in listing 7.43 defines the transition for parallel gateways by refining the rule `DataBasedGatewayTransition : DATA_BASED_GATEWAYS` (see listing 7.42) and specifying the `MergeBehavior` to be the `ParallelMergeBehavior` and the `SplitBehavior` to be the `ParallelSplitBehavior`.

---

**Listing 7.43** rule `ParallelGatewayTransition : PARALLEL_GATEWAYS`

---

```

1 rule ParallelGatewayTransition(gateway) =
2   DataBasedGatewayTransition(gateway) where
3     GatewayBehavior(gateway) = InputBehavior(gateway) where
4       MergeBehavior(gateway) = ParallelMergeBehavior(gateway)
5
6   GatewayOperation(gateway, instances) =
7     OutputBehavior(gateway, instances) where
8       SplitBehavior(gateway, instance) =
9         ParallelSplitBehavior(gateway, instance)

```

---

The rule `ExclusiveGatewayTransition : EXCLUSIVE_GATEWAYS` shown in listing 7.44 defines the transition for exclusive gateways by refining the rule `DataBasedGatewayTransition : DATA_BASED_GATEWAYS` (see listing 7.42) and specifying the `MergeBehavior` to be the `ExclusiveMergeBehavior` and the `SplitBehavior` to be the `ExclusiveSplitBehavior`.

The rule `InclusiveGatewayTransition : INCLUSIVE_GATEWAYS` shown in listing 7.45 defines the transition for inclusive gateways by refining the rule `DataBasedGatewayTransition : DATA_BASED_GATEWAYS` (see listing 7.42) and specifying the `MergeBehavior` to be the `InclusiveMergeBehavior` and the `SplitBehavior` to be the `InclusiveSplitBehavior`.

The rule `ComplexGatewayTransition : COMPLEX_GATEWAYS` shown in listing 7.46 defines the transition for complex gateways by refining the rule `DataBasedGatewayTransition : DATA_BASED_GATEWAYS` (see listing 7.42) and specifying the `MergeBehavior` to be the `ComplexMergeBehavior` and the `SplitBehavior` to be the `InclusiveSplitBehavior`. The one difference of

---

**Listing 7.44** rule ExclusiveGatewayTransition : EXCLUSIVE\_GATEWAYS

---

```
1 rule ExclusiveGatewayTransition(gateway) =  
2   DataBasedGatewayTransition(gateway) where  
3     GatewayBehavior(gateway) = InputBehavior(gateway) where  
4       MergeBehavior(gateway) = ExclusiveMergeBehavior(gateway)  
5  
6   GatewayOperation(gateway, instances) =  
7     OutputBehavior(gateway, instances) where  
8       SplitBehavior(gateway, instance) =  
9         ExclusiveSplitBehavior(gateway, instance)
```

---

---

**Listing 7.45** rule InclusiveGatewayTransition : INCLUSIVE\_GATEWAYS

---

```
1 rule InclusiveGatewayTransition(gateway) =  
2   DataBasedGatewayTransition(gateway) where  
3     GatewayBehavior(gateway) = InputBehavior(gateway) where  
4       MergeBehavior(gateway) = InclusiveMergeBehavior(gateway)  
5  
6   GatewayOperation(gateway, instances) =  
7     OutputBehavior(gateway, instances) where  
8       SplitBehavior(gateway, instance) =  
9         InclusiveSplitBehavior(gateway, instance)
```

---

the `SplitBehavior` of complex gateway and inclusive gateway is that the `gate-ConditionExpression` may additionally guard the current state of the complex gateway [1, tab. 13.5].

---

**Listing 7.46** rule ComplexGatewayTransition : COMPLEX\_GATEWAYS

---

```
1 rule ComplexGatewayTransition(gateway) =  
2   DataBasedGatewayTransition(gateway) where  
3     GatewayBehavior(gateway) = InputBehavior(gateway) where  
4       MergeBehavior(gateway) = ComplexMergeBehavior(gateway)  
5  
6   GatewayOperation(gateway, instances) =  
7     OutputBehavior(gateway, instances) where  
8       SplitBehavior(gateway, instance) =  
9         InclusiveSplitBehavior(gateway, instance)
```

---

Herewith we defined the formal semantics for the BPMN meta-model discussed in previous chapters. We leave further refinements of more specific flow node types for some future work.

## 7.7 Removed parts from the original BPMN specification

For the sake of completeness we list here the functions and rules, originating from the BPMN meta-model and discussed in previous chapters to be left out from the ground model.

The static function `conditionExpression` shown in signature 7.33 is an optional boolean expression that acts as a gating condition in [1, tab. 8.51]. This was removed in the refined ground model due to the `GateBehavior`.

```
static conditionExpression : SEQUENCE_FLOWS →  
EXPRESSIONS [7.33]
```

The static function `isImmediate` shown in signature 7.34 represents an optional attribute of a sequence flow (see [1, tab. 8.51] for more details). Since this attribute must be always `true` with executable models, which are the only relevant models within this thesis, this function was removed from the refined ground model.

```
static isImmediate : SEQUENCE_FLOWS → BOOLEAN [7.34]
```

The `defaultSequenceFlow` shown in signature 7.35 represents the optional parameter of the `Activity` meta-model class identifying the default flow in case none of the other conditional flow `conditionExpressions` evaluate to `true` [1, tab. 10.3]. Since this behavior is defined duplicate for inclusive, exclusive and complex gateways and we model such compound behavior using reusable blocks as discussed in chapter 6, we do not need this property for activities any more.

```
static defaultSequenceFlow : ACTIVITIES → SEQUENCE-  
_FLOWS [7.35]
```

The static function `completionQuantity` shown in signature 7.36 has the default value of 1. The value must not be less than 1. This attribute defines the number of tokens that must be generated from the activity and sent down to any outgoing sequence flow (assuming any sequence flow conditions are satisfied) [1, tab. 10.3]. Note that since any value for the attribute that is greater than 1 should be according to [1] used with caution we do not include this into the ground model at this

moment. This functionality should be further revised and refined in case it will be incorporated in some future work.

```
static completionQuantity : ACTIVITIES → NATURAL-
_NUMBERS [7.36]
```

The static function `eventDefinitions` shown in signature 7.37 defines the event triggers expected for a catch event or throw event [1, tab. 10.82, 10.83]. They are originally defined as two separate attributes, one for catch events and one for throw events. But since every event has to be either a catch event or throw event this attribute can be defined in the *Event* meta-model class to avoid duplication. Due to naming conventions this attribute was renamed and is now available via the `triggers` function (see signature 7.23).

```
static eventDefinitions : EVENTS → SET(TRIGGERS) [7.37]
```

The attribute `eventDefinitionRefs` [1, tab. 10.82, 10.83] represented by the static function shown in signature 7.38 is similar to static function `eventDefinitions` (see signature 7.37). The only difference between those two functions is that the `eventDefinitionRefs` references reusable triggers, while `eventDefinitions` contains triggers that are only valid inside the current event.

```
static eventDefinitionRefs : EVENTS → SET(TRIGGERS) [7.38]
```

The following five static functions: the reference : `SIGNAL_TRIGGERS` → `SIGNALS` shown in signature 7.39, the reference : `MESSAGE_TRIGGERS` → `SIGNALS` shown in signature 7.40, the reference : `EXCEPTION_TRIGGERS` → `SIGNALS` shown in signature 7.41, the reference : `ERROR_TRIGGERS` → `SIGNALS` shown in signature 7.42, and the reference : `ESCALATION_TRIGGERS` → `SIGNALS` shown in signature 7.43 define a link between the signal trigger, the message trigger, the exception trigger, the error trigger, and the escalation trigger respectively, and the payload [1, fig. 10.93]. This link has been removed due to refinements since notifications do not exist before they are thrown and matched to a concrete event node using a certain trigger.

```
static reference : SIGNAL_TRIGGERS → SIGNALS [7.39]
```

```
static reference : MESSAGE_TRIGGERS → MESSAGES [7.40]
```

```
static reference : EXCEPTION_TRIGGERS → EXCEPTIONS [7.41]
```

```
static reference : ERROR_TRIGGERS → ERRORS [7.42]
```

```
static reference : ESCALATION_TRIGGERS → ESCALATIONS [7.43]
```

The attribute represented by the static function `gatewayDirection : GATEWAYS → GATEWAY_DIRECTION` can gain the following values [1, tab. 8.46]: *Unspecified*, *Converging*, *Diverging*, or *Mixed*. While a *Converging* gateway has to have at least one, but typically multiple, incoming sequence flows but exactly one outgoing sequence flow, and the other way around for a *Diverging* gateway - exactly one incoming sequence flow and one, but typically multiple, outgoing sequence flows, we argue that in case of exactly one incoming and exactly one outgoing sequence flow a gateway is neither a *Converging* nor *Diverging* gateway. We argue that the relevant values from the ones mentioned are only *Converging*, *Diverging*, and *Mixed*. The *Unspecified* can then only occur in case of exactly one incoming and one outgoing sequence flow. This case we call *PassThrough*, which can be used to replace a conditional flow.

```
static gatewayDirection : GATEWAYS → GATEWAY-  
_DIRECTIONS [7.44]
```

The `eventGatewayType` shown in signature 7.45 in the BPMN specification [1, tab. 10.127] determines the behavior of the gateway when used to instantiate a process. Due to the refinements in this work this attribute was removed.

```
static eventGatewayType : EVENT_BASED_GATEWAYS → EVENT-  
_GATEWAY_TYPES [7.45]
```

The static function `instantiate` shown in signature 7.46 determines, whether a process should be instantiated using the given event-based gateway. This functionality was refined and this function was removed from the refined ground model.

```
static instantiate : EVENT_BASED_GATEWAYS → BOOLEAN [7.46]
```

In the next part we will concentrate on certain refinements further down in the vertical direction targeting a WFL.



*A debugged program is one for which you have not yet found the conditions that make it fail.*

— Jerry Ogdin

# Part III

## Workflow Interpreter of the Workflow Engine

In this part of the thesis we propose a formal specification of the control flow in Business Process Model and Notation (BPMN) Workflow Engine (WFE) [93]. We call the part of BPMN WFE, which is responsible for the control flow of a BPMN process only the BPMN Workflow Interpreter (WFI). Nowadays there are already some existing BPMN WFEs on the market. We can list for example: JBoss RedHat2012 [29], Activity ([94]), Bonita Execution Engine (<http://www.bonitasoft.com>), Route (<http://ruote.rubyforge.org/>), or Enhydra Shark (<http://shark.enhydra.org/>). The preference for evaluation is put on those WFE, which do support the current 2.0 version of the BPMN standard [1]. The resulting Abstract State Machine (ASM) ground model [2] may be a starting point for the future implementations of a WFE. In any case of contradictions between the BPMN 2.0 standard and a BPMN WFE, the BPMN standard will be endorsed.

This leads us to one of the goals of this thesis, where in some future work, the intended BPMN WFI ASM ground model may be used to prove correct behavior of a concrete implementation of a BPMN WFI, if such an implementation is based on

this ASM ground model. Or on the other hand, to show an incorrect behavior of an implementation of a BPMN WFI, where the concrete implementation may be out of the scope of the BPMN 2.0 standard. For this purpose we could use for example model checking techniques [95].

A similar approach of formalization of an existing system was described in [53], where Java and the Java Virtual Machine (JVM) were specified using ASMs. To make a relation between [53] and our work, we can look at the BPMN 2.0 standard as a Java program described in the 1<sup>st</sup> part of [53] and at the BPMN WFI as a JVM, or at least its part, in the 2<sup>nd</sup> part of [53]. In [53] Java and JVM were decomposed into 5 stages: imperative, static class features, object-oriented features, exception handling, and concurrent threads. Every stage was a refinement (read: extension) of the previous stage. Each stage of a Java program has a corresponding stage of a JVM. In this sense the BPMN is decomposed into the following stages: control flow, message flow, Data Flow, Exception Flow, etc. with the `FlowNodeBehaviour : FLOW_NODES` rule in [17] or with the `WorkflowTransition : FLOW_NODES` rule in [16]. In both cases the mentioned rules fire `DataOperation : FLOW_NODES`, `ControlOperation : FLOW_NODES`, `EventOperation : FLOW_NODES` and `ResourceOperation : FLOW_NODES` if certain conditions are fulfilled, each representing a certain decomposition stage<sup>1</sup>.

In such a sense we want to specify the control flow part of a BPMN WFE formally using the mentioned ASM method to make it possible, in some future work, to bring together those two parts: the BPMN 2.0 standard ASM ground model [16, 15, 17] and the BPMN WFE ASM ground model, similarly as in [53].

---

<sup>1</sup>Note that in this work we intentionally merged the control and event related conditions and operations as discussed in section 6.7

*Life is NP-hard, and then you die.*

— Dave Cock

## Chapter 8

# Targeting the WFI

In this chapter we will further refine the ground model from the previous chapters and define abstract derived functions and rules, which were left abstract on the business level of the ground model. The motivation for this chapter is to fill the gap between the abstract BPMN ASM ground model defined in chapter 7, which is based on [17, 96] and the WFI. The basic idea of how a WFI works can be seen in [96]: a rule `WorkflowTransitionInterpreter : PROCESSES`, which fire the rule `WorkflowTransition : FLOW_NODES`. This abstract rule handles the traversal of a token through all instances of all processes deployed to the WFE, which the WFI is the core part of. The rule `WorkflowTransition : FLOW_NODES` has been refined in chapter 7, where [17] was taken as a starting point, for all concrete flow node types such as activities, gateways and events. The communication between the process run and the WFI is refined in this chapter by defining the rules and locations left abstract in [17] concerning the communication between processes using messages or signals.

Starting with defining the execution context tree in section 8.1, where the root of that tree is the static context, present only once. Its immediate children are root contexts created for each running top-level process as soon as such a top-level process is started. The rest of the context tree is formed by sub contexts, which are created for every new activity instance, such as sub-process or call activity. After building the context tree, notifications defined in section 8.2 can carry triggers to their destination (events) through it. These notifications are ordered according to the `occurrenceTimeOfNotification`. *Notifications* can be assigned to concrete flow nodes. Implicit triggers, defined in section 8.3, that are triggered when certain conditions occur in the workflow, automatically throw corresponding notifications in such a case. Inter-process communication is possible in [1] using messages and signals. For those we define the creation of the corresponding notifications in section 8.3.1. Then, in section 8.3.2 and section 8.3.3, the publication and the propagation forwarding concepts [1] will be refined. The publication concept forwards the notifications down the context tree while the propagation concept forwards them up the context tree. In section 8.4 we briefly sketch the deployment concept of a WFI, which is further left for some future work. In section 8.5 we define the upstream to-

ken discussed in 6.3 and introduce an algorithm to compute such upstream tokens. Finally, in section 8.6 we transit the ground model to the technical level.

## 8.1 Contexts

In section 5.2.3 we introduced the notification concept as a similar concept to the token concept by carrying a notification, which should be caught by a catch event or propagated up to the WFI depending on the forwarding concept. A token traverses through sequence flows, but there is no definition of any medium a notification traverses through. This was not needed on the business level, but a WFI needs it. We call this medium a context. A context is simply stated, a wrapper around an instance holding additional information, created and managed by the WFI. Also the environment communicates with the WFI through contexts. There are three types of contexts but only two of them allow to be accessed by the environment.

**Static context** is the first one and exists only once. It is created as soon as the WFE is started. All existing deployments expose their defined Top-level start events in the static context, which are those with no defined trigger or with Message trigger, Timer trigger, Conditional trigger or Signal trigger [1], to the Static context waiting for a corresponding notification to be fired (see section 8.2 for additional details about notifications). The environment puts such a notification to the `monitored notifications → PRIORITY_QUEUE` and as soon as such a notification arrives, the Instance manager will instantiate the corresponding Process and create a Root context for it. Also the function `notifications` will return the notifications ordered by the time of their occurrence from the oldest ones to the newest ones. For that purpose the `monitored occurrenceTime : NOTIFICATIONS → TIME` is used.

**Root context** is a context created for each new process instance. The environment can communicate with this process by sending notification to an existing intermediate event or by setting `to true` as soon as the corresponding task was finished.

**Sub context** is a context created for every new activity instance inside a running process deployment. This context is not visible to the environment but may populate some uncaught events (unconsumed notifications) to the parent context. In the same sense, uncaught events (unconsumed notifications) sent to the root context may be populated to the sub context. Also waiting tasks are populated to the Root context using the `derived waitingTasks : CONTEXTS → SET(-TASKS)` function shown in listing 8.1.

The function `parentContext` shown in signature 8.1 returns a parent context of the given context. If the given context is the static context of the running WFE than this function returns `undef`.

---

**Listing 8.1** `derived waitingTasks : CONTEXTS → SET(TASKS)`

---

```
1 derived waitingTasks(context) =  
2   return res in  
3     let parentI ← instance(context) in  
4       local subC ← subContexts(context),  
5         waiting ← { t | t ∈ flowNodes(instantiatingFlowNode(parentI))  
6           ∧ t ∈ TASKS  
7           ∧ { i | i ∈ activeInstances(t)  
8             ∧ parentInstance(i) = parentI  
9             ∧ completed(t, i) = ⊥ } } in  
10  
11     while subC ≠ ∅ do  
12       choose childContext ∈ subC do  
13         waiting ← waiting ∪ waitingTasks(childContext)  
14       remove childContext from subC  
15  
16   res ← waiting
```

---

monitored parentContext : CONTEXTS → CONTEXTS	[8.1]
---	-------

The derived function `subContexts` shown in listing 8.2 computes all sub contexts of the given context.

---

**Listing 8.2** `derived subContexts : CONTEXTS → SET(CONTEXTS)`

---

```
1 derived subContexts(parent) =  
2   { context | context ∈ CONTEXTS ∧ parentContext(context) = parent }
```

---

## 8.2 Notifications

Notifications, as defined in section 5.2.3, fire an event containing a concrete trigger see [1, sec. 10.4.5]. A notification has therefore to specify:

- the context of the notification stored in signature 8.2,

shared context : NOTIFICATIONS → CONTEXTS	[8.2]
---	-------

- the flow node of the notification in signature 8.3, and

shared flowNode : NOTIFICATIONS → FLOW_NODES	[8.3]
--	-------

- the time of occurrence shown in signature 8.4.

$$\text{monitored occurrenceTime} : \text{NOTIFICATIONS} \rightarrow \text{TIME} \quad [8.4]$$

Since a notification can be created by both, the environment and the WFI machine, the first two functions are shared. The `occurrenceTime` is monitored since the value is generated automatically for each notification by some arbitrary clock responsible machine, which we do not define at this point. Notifications are one way of how the environment can communicate with a context of the WFE.

### 8.3 Implicit throw events

The concept of implicitly thrown events (defined in [1, sec. 10.4.1]) is discussed in this section to enable the WFI control implicit events. The rule `ThrowImplicitNotification` shown in listing 8.3, is responsible for observing conditional and timer trigger and throw corresponding notifications if their conditions were met. The assumption made here is that every timer trigger can be generalized to a conditional trigger and the attributes: `timeDate`, `timeCycle` and `timeDuration` [1, tab. 10.101] can be expressed by a condition [1, tab. 10.95].

---

**Listing 8.3** rule `ThrowImplicitNotifications`

---

```

1 rule ThrowImplicitNotifications =
2   ∀ event ∈ CATCH_EVENTS do
3     ∀ trigger ∈ triggers(event) with
4       trigger ∈ CONDITIONAL_TRIGGERS do
5       ∀ i ∈ INSTANCES with evaluate(conditionExpression(trigger), i) = T do
6         choose ctx ∈ CONTEXTS with instance(ctx) = i do
7           let notification ← new NOTIFICATIONS in
8             context(notification) ← ctx
9             flowNode(notification) ← event

```

---

As the `shared flowNode : NOTIFICATIONS → FLOW_NODES` is defined with the creation of the notification, this forwarding concept is referred to as “direct resolution” in [1].

#### 8.3.1 Message and signal pools

For the purpose of inter process communication the BPMN standard defines messages and signals [1]. The main difference is that a message specifies a target but a signal broadcasts to all signal catch events. The second most significant difference is that signals just trigger the corresponding catch events but messages usually carry more complex content, i.e., `payloadOfMessageNotification`, and also may call a service operation [1, sec. 8.4.3, fig. 8.30, 10.89].

The source and the target event of a message are linked by a message flow defined in collaboration [1, ch. 9], but this is out of the execution scope and thus not relevant to this thesis. First, collaboration is not required for process execution conformance nor for Business Process Execution Language (BPEL) process execution conformance [1, p. 109]. Second, implementation of message flows will require loading all communicating processes into one WFI, which is not always possible, since different processes may be running on different WFE and still be communicating. For this purpose we define a `messagePool` and a `signalPool`. Messages or signals arriving from the environment are converted to notifications as defined in the rule `ProcessMessagePool`, shown in listing 8.4, and similarly for signals in the rule `ProcessSignalPool`, shown in listing 8.5.

---

**Listing 8.4** rule `ProcessMessagePool`

---

```

1 rule ProcessMessagePool =
2   ∀ message ∈ messagePool do
3     let notification ← new MESSAGE_NOTIFICATIONS in
4       context(notification) ← "StaticContext"
5       name(notification) ← name(message)
6       payload(notification) ← payload(message)
7
8   messagePool ← ∅

```

---



---

**Listing 8.5** rule `ProcessSignalPool`

---

```

1 rule ProcessSignalPool =
2   ∀ signal ∈ signalPool do
3     let notification ← new SIGNAL_NOTIFICATIONS in
4       context(notification) ← "StaticContext"
5       name(notification) ← name(signal)
6
7   signalPool ← ∅

```

---

### 8.3.2 Publication resolution

The publication resolution forwarding concept defined in [1] and discussed in section 5.2.3 applies for message and signal events. The creation of notifications for such events coming from the environment is shown in section 8.3.1. The publication of those notifications is defined in listing 8.6.

As communicating processes can be running on different WFE the proposal is to define the message flow as a matching concept based on the name of the message (see [1, fig. 10.89]). After a message event with the corresponding name is found, it will be fired and the notification will be consumed. Similarly for signals [1, fig. 10.93], but with the exception that the corresponding notification will not be consumed, which allows multiple signal catch events to catch the signal. For this purpose we de-

defined the controlled name :  $\text{SIGNAL\_NOTIFICATIONS} \rightarrow \text{STRINGS}$  function.

---

**Listing 8.6** rule PublishNotification :  $\text{SIGNAL\_NOTIFICATIONS}$

---

```

1 rule PublishNotification(notification) =
2   //ForwardNotification(notification) where
3   local c ← context(notification) in
4     ∀ event ∈ CATCH_EVENTS do
5       ∀ trigger ∈ triggers(event) ∩ SIGNAL_TRIGGERS with
6         name(trigger) = name(notification) do
7         if trigger ∉ MESSAGE_TRIGGERS then
8           let duplicate ← Clone(notification) in
9             flowNode(duplicate) ← event
10
11       else
12         flowNode(notification) ← event
13
14     ∀ task ∈ RECEIVE_TASKS do
15       flowNode(notification) ← task
16
17   / Publish only yet unassigned notifications: /
18   if flowNode(notification) = undef then
19     ∀ child ∈ CONTEXTS with parentContext(child) = c do
20       let duplicate ← Clone(notification) in
21         context(duplicate) ← child
22
23   remove notification from notifications // lifetime ends

```

---

The abstract rule Clone :  $\text{NOTIFICATION} \rightarrow \text{NOTIFICATIONS}$  duplicates the given notification with preserving the original monitored occurrenceTime :  $\text{NOTIFICATIONS} \rightarrow \text{TIME}$ . Creating a new notification with the new construct would generate new timestamp.

### 8.3.3 Propagation resolution

Additionally, events from running process instances may be propagated up to their innermost context containing an event, which can catch them [1, sec. 10.4.1]. If no such event is defined, those notifications may be propagated up to their root context or to the common static context, e.g., error, escalation.

The corresponding notifications are created in the refined rule Throw :  $\text{EXCEPTION\_TRIGGERS} \times \text{INSTANCES}$  [17] shown in listing 8.7.

If such an error notification is not caught by any enclosing activity, the process instance will terminate in case of an error trigger [1, tab. 10.88]. In case of other escalation triggers nothing will happen as this is defined as the common behavior in [1] and may be refined in a concrete implementation, as shown in listing 8.8.



---

**Listing 8.7** rule Throw : EXCEPTION\_TRIGGERS  $\times$  INSTANCES

---

```
1 rule Throw(trigger, inst) =  
2   let notification in  
3     if trigger  $\in$  ERROR_TRIGGERS then  
4       notification  $\leftarrow$  new ERROR_NOTIFICATIONS  
5     else  
6       notification  $\leftarrow$  new ESCALATION_NOTIFICATIONS  
7  
8     choose ctx  $\in$  CONTEXTS with instance(ctx) = inst do  
9       context(notification)  $\leftarrow$  ctx  
10      code(notification)  $\leftarrow$  code(trigger)  
11      name(notification)  $\leftarrow$  name(trigger)
```

---

---

**Listing 8.8** rule PropagateNotification : EXCEPTION\_NOTIFICATIONS

---

```
1 rule PropagateNotification(notification) =  
2   //ForwardNotification(notification) where  
3     choose boundary  $\in$  BOUNDARY_EVENTS with  
4       attachedTo(boundary) =  
5         instantiatingFlowNode(instance(context(notification)))  
6        $\wedge$  ( $\exists$  trigger  $\in$  (triggers(boundary)  $\cap$  EXCEPTION_TRIGGERS) with  
7         code(trigger) = code(notification)) do  
8       flowNode(notification)  $\leftarrow$  boundary  
9  
10    if flowNode(notification) = undef then  
11      let parent  $\leftarrow$  parentContext(context(notification)) in  
12        if parent  $\in$  SUB_CONTEXTS then  
13          context(notification)  $\leftarrow$  parent  
14        else if parent  $\in$  ROOT_CONTEXTS  
15           $\wedge$  notification  $\in$  ERROR_NOTIFICATIONS then  
16            Terminate  
17  
18    // If no parent, we already terminated and are in static context
```

---

## 8.4 Deployments and deployment manager

Deployments are BPMN Process Diagrams (PDs) loaded most commonly from an Extensible Markup Language (XML) file to the WFE, where they are stored in the corresponding binary form and can be exported to an XML file back again, since the relation between the XML based and binary based version is 1:1 (see [1] for XML definition of BPMN diagrams). Any additional information added to a deployment are runtime specifics and are irrelevant for the PD model.

A new deployment can be added by adding it to the location monitored `newDeploymentsRequestedByEnvironment` : `Deployment` which is observed by the rule `HandleNewDeployments` in the deployment manager shown in listing 8.9. A new deployment is added only if it is not already present. Adding the same BPMN model described in XML, will not result two distinct deployments in the deployment manager, unless the original XML description has been changed.

---

**Listing 8.9** rule `HandleNewDeployments`

---

```
1 rule HandleNewDeployments =
2   ∀ deployment ∈ newDeploymentsRequestedByEnvironment do
3     ∀ expressionLanguage ∈ expressionLanguages(deployment) do
4       CheckExpressionLanguage(expressionLanguage)
5
6   ∀ trigger ∈ triggers(deployment) do
7     CheckTrigger(trigger)
8
9   if ¬ deployment ∈ deployments then
10    add deployment to deployments
11
12  remove deployment from newDeploymentsRequestedByEnvironment
```

---

The rule `Dispatch` shown in listing 8.10 is a modified `CreateInstance` responsible for dispatching – creating an instance and throwing a given trigger – a top-level process.

---

**Listing 8.10** rule `Dispatch` : `PROCESSES` × `TRIGGERS`

---

```
1 rule Dispatch(process, trigger) =
2   let instance ← new INSTANCES in
3     add instance to activeInstances(process)
4     instantiatingProcess(instance) ← process
5     lifeCycleState(instance, process) ← "Ready"
6     Throw(trigger, instance)
```

---

We sketched the deployment concept in this section to distinguish between, deployed processes and instantiated processes. However, we leave the refinements of this concept to some future work.

## 8.5 Upstream token

Upstream tokens were already discussed in section 6.3 where they were used as an abstract rule. An upstream token of a flow node, defined in listing 8.12, is a token in a sequence flow if there is a path starting with that sequence flow and reaching the given flow node with all other sequence flows building that path not having a token, if such a sequence flow is not directly connected to the given flow node and there is no alternative path from that sequence flow to the given flow node reaching a directly connected sequence flow to the given flow node having a token. Upstream tokens are needed for flow nodes such as inclusive and complex gateways. This is defined using inhibiting and anti-inhibiting paths [60] as a token, which has an inhibiting path but no anti-inhibiting path to the corresponding flow node. Such upstream tokens are used as an activation condition of an inclusive gateway [1, tab. 13.3], or as a reset condition of a complex gateway [1, tab. 13.5].

In this section we propose an algorithm as a refinement of the rule - UpstreamTokens :  $\text{FLOW\_NODES} \times \text{SET}(\text{SEQUENCE\_FLOWS}) \times \text{SET}(\text{-TOKENS}) \rightarrow \text{SET}(\text{TOKENS})$  [15, 17], which will identify all relevant sequence flows for any given flow node in a process diagram and color them based on their directly incoming sequence flows to the chosen flow node. This computation and coloring is made on an oriented cyclic graph represented by an ordered pair  $\mathcal{G}(\mathcal{N}, \mathcal{E})$ . The  $\mathcal{G}(\mathcal{N}, \mathcal{E})$  can be obtained by converting all flow nodes of the process diagram to nodes  $\mathcal{N}$  of  $\mathcal{G}$  and all sequence flows to oriented edges  $\mathcal{E}$  with opposite orientation to the original sequence flows of the process diagram.

**Definition 1** (Graph transformation). *Let  $\mathcal{G}_d(\mathcal{N}, \mathcal{E}_{sf})$  be a process diagram,  $\mathcal{N}$  be a set of all flow nodes in  $\mathcal{G}_d$  and  $\mathcal{E}_{sf}$  be set of all sequence flows in  $\mathcal{G}_d$ . We construct an oriented graph  $\mathcal{G}(\mathcal{N}, \mathcal{E})$  where  $\forall e \in \mathcal{E} \exists! e_{sf} \in \mathcal{E}_{sf} (e = \{(n_0, n_1) \in \mathcal{N}\} \wedge e_{sf} = \{(n_1, n_0) \in \mathcal{N}\})$ .*

The next step is to color the edges of the graph  $\mathcal{G}$  for every flow node that requires such coloring for its activation, e.g., inclusive or complex gateways, using the modified Dijkstra's algorithm [59] shown in listing 8.11.

**Definition 2** (Colored graph). *A colored graph is a tuple  $\mathcal{G}_K = (\mathcal{N}, \mathcal{E}, \kappa, \delta, \mathcal{C}_{\mathcal{N}}, \mathcal{C}_{\mathcal{E}})$ , where:*

- $\mathcal{N}$  is a set of nodes same as in  $\mathcal{G}_d$  or  $\mathcal{G}$ ,
- $\mathcal{E} : \mathcal{N} \times \mathcal{N}$  is a set of edges after transforming  $\mathcal{G}_d$  to  $\mathcal{G}$ , where the first parameter represents a node the edge is outgoing from and the second parameter stands for the node the edge is incoming to<sup>1</sup>,
- $\kappa : \mathcal{N} \times \mathcal{N} \rightarrow \text{Boolean}$  is a function capturing closeness of the node related to a node, a color calculation is done for,
- $\delta : \mathcal{N} \times \mathcal{N} \rightarrow \text{Integer}$  is a function capturing the distance between two nodes,

<sup>1</sup>Note that the direction of edges  $e \in \mathcal{E}$  is opposite to the direction of sequence flows  $s \in \mathcal{E}_{sf}$

- $\mathcal{C}_{\mathcal{N}} : \mathcal{N} \times \mathcal{N} \rightarrow \text{Color}$  is a function capturing a node color related to a node, a color calculation is done for,
- $\mathcal{C}_{\mathcal{E}} : \mathcal{N} \times \mathcal{E} \rightarrow \text{Color}$  is a function capturing an edge color related to a node, a color calculation is done for.

The first node parameter of the functions  $\kappa$ ,  $\delta$ ,  $\mathcal{C}_{\mathcal{N}}$  and  $\mathcal{C}_{\mathcal{E}}$  represents the node for which a color calculation is done. The second parameter of those functions represents the relevant node or edge on the incoming inhibiting or anti-inhibiting paths of the node passed to those functions in the first parameter.

**Definition 3** (Upstream token). A flow node represented by  $n \in \mathcal{N}$  in  $\mathcal{G}_k$  has an upstream token if:

- there is a token in a relevant (for that node  $n$  colored with one or more colors) edge  $e \in \mathcal{E}$ , which has a token
- and there is no directly outgoing edge from node  $n$  which has a token and is colored with one of the colors  $e$  is also colored with

### 8.5.1 Work of the algorithm

The `ColorProcessGraph` rule shown in 8.11 finds a shortest path from the root node of  $\mathcal{G}_k$  (representing an inclusive or complex gateway in  $\mathcal{G}_d$ ) given as the only parameter to any reachable node in the directed graph  $\mathcal{G}_k$  in the way the original Dijkstra's algorithm [59] was designed. Additionally the algorithm colors visited nodes and tested edges, even those tested edges which are not further used for the search (i.e., incoming edges of closed nodes  $n \in \mathcal{N}$  indicated by  $\kappa(\text{root}, n) = \top$ ). The algorithm starts with coloring each of the directly outgoing edges from the root node with a distinct color. Those colors are then populated through  $\mathcal{G}_k$  till its leafs (usually representing start events in  $\mathcal{G}_d$ ). If the algorithm visits a node which has more than one incoming edge in  $\mathcal{G}_k$  (representing a splitting gateway in  $\mathcal{G}_d$ ) for the first time (i.e., such node  $n \in \mathcal{N}$  is not closed yet:  $\kappa(\text{root}, n) = \perp$ ), it will additionally color this node with a distinct `limpid` color. Such `limpid` color will be further populated with the other colors such a node is colored with. A color calculation for a root node (represented by a run of `ColorProcessGraph`) will never color two nodes in  $\mathcal{G}_k$  having more than one incoming edge with the same `limpid` color. A `limpid` color is an indicator for other alternative paths of the concrete node to populate their own non-`limpid` colors. Every time such a node will be tested by the algorithm again (i.e., such a node is already closed), all non-`limpid` colors, of the tested alternative edge will be populated to all nodes and edges in  $\mathcal{G}_k$  which are colored with the `limpid` color of the tested node.

All `limpid` colors are only visible locally inside the run of the `ColorProcessGraph`. The resulting set of colors obtained from both color functions ( $\mathcal{C}_{\mathcal{N}}$ ,  $\mathcal{C}_{\mathcal{E}}$ ) will not contain any `limpid` colors.

---

**Listing 8.11** rule ColorProcessGraph : N
 

---

```

1 rule ColorProcessGraph(root) =
2    $\forall n \in \mathcal{N}$  do
3      $\kappa(\text{root}, n) \leftarrow \perp$ 
4      $\delta(\text{root}, n) \leftarrow \infty$ 
5      $\mathcal{C}_{\mathcal{N}}(\text{root}, n) \leftarrow \emptyset$ 
6
7    $\forall e \in \mathcal{E}$  do  $\mathcal{C}_{\mathcal{E}}(\text{root}, e) \leftarrow \emptyset$ 
8    $\delta(\text{root}, \text{root}) \leftarrow 0$ 
9   local set  $\leftarrow \{\text{root}\}$ , color in
10    while |set| > 0 do
11      choose node  $\in \{n \mid n \in \text{set}\}$ 
12         $\wedge \nexists m \in \text{set} : \delta(\text{root}, n) > \delta(\text{root}, m)$  holds do
13
14      remove node from set
15       $\kappa(\text{root}, \text{node}) \leftarrow \top$ 
16
17      foreach next  $\in \mathcal{N}$  with  $\mathcal{E}(\text{node}, \text{next}) \neq \text{undef} \wedge \text{next} \neq \text{root}$  do
18        if  $\delta(\text{root}, \text{node}) + 1 < \delta(\text{root}, \text{next})$ 
19           $\wedge \neg \kappa(\text{root}, \text{next})$  then
20
21           $\delta(\text{root}, \text{next}) \leftarrow \delta(\text{root}, \text{node}) + 1$ 
22          add next to set
23
24          if  $\exists n \in \mathcal{N}$  with  $\mathcal{E}(n, \text{next}) \neq \text{undef} \wedge n \neq \text{node}$  then
25            add nextLimpid to  $\mathcal{C}_{\mathcal{N}}(\text{root}, \text{node})$ 
26
27      if node = root then
28        color  $\leftarrow$  nextColor
29        add color to  $\mathcal{C}_{\mathcal{N}}(\text{root}, \text{next})$ 
30        add color to  $\mathcal{C}_{\mathcal{E}}(\text{root}, \mathcal{E}(\text{node}, \text{next}))$ 
31      else
32        foreach color  $\in \mathcal{C}_{\mathcal{N}}(\text{root}, \text{node})$  do
33          if ( $\nexists c \in \mathcal{C}_{\mathcal{N}}(\text{root}, \text{next}) : \text{isLimpid}(c)$  holds)
34             $\vee \neg \text{isLimpid}(\text{color})$  then
35              add color to  $\mathcal{C}_{\mathcal{N}}(\text{root}, \text{next})$ 
36
37        add color to  $\mathcal{C}_{\mathcal{E}}(\text{root}, \mathcal{E}(\text{node}, \text{next}))$ 
38
39      foreach limpid  $\in \mathcal{C}_{\mathcal{N}}(\text{root}, \text{next})$  with  $\text{isLimpid}(\text{limpid})$  do
40         $\forall n \in \mathcal{N}$  with limpid  $\in \mathcal{C}_{\mathcal{N}}(\text{root}, n)$  do
41          add color to  $\mathcal{C}_{\mathcal{N}}(\text{root}, n)$ 
42
43       $\forall e \in \mathcal{E}$  with limpid  $\in \mathcal{C}_{\mathcal{E}}(\text{root}, e)$  do
44        add color to  $\mathcal{C}_{\mathcal{E}}(\text{root}, e)$ 

```

---

### 8.5.2 Enableness test

The enableness of an inclusive gateway is defined as:

**Theorem 1** (Enableness of inclusive gateway). *The Inclusive Gateway is enabled if [1, 17, 60]:*

- At least one incoming sequence flow has at least one token and
- there is no upstream token, meaning:
  - a token that has an inhibiting path,
  - but no anti-inhibiting path

---

**Listing 8.12** rule UpstreamTokens : FLOW\_NODES  $\times$  SET(SEQUENCE-  
\_FLOWS)  $\times$  SET(TOKENS)  $\rightarrow$  SET(TOKENS)

---

```

1 rule UpstreamTokens(flowNode, sequenceFlows, tokens) =
2   // colors to ignore
3   let ignore  $\leftarrow$  { c |  $\exists e \in$  sequenceFlows with
4     |tokensInSequenceFlow(e)  $\cap$  tokens| > 0 holds
5      $\wedge c \in \mathcal{C}_{\mathcal{E}}(\text{flowNode}, e)$  } in
6   // all relevant sequence flows in the diagram
7   let relevant  $\leftarrow$  { e | e  $\in$  EDGES
8      $\wedge \mathcal{C}_{\mathcal{E}}(\text{flowNode}, e) \setminus \text{ignore} \neq \emptyset$  } in
9
10  return res in
11    res  $\leftarrow$  { t |  $\exists e \in$  relevant
12       $\wedge \exists t \in$  tokensInSequenceFlow(e) with
13        instance(t) = instance(tokens)
14         $\wedge (\mathcal{C}_{\mathcal{E}}(\text{flowNode}, e) \setminus \text{ignore}) \neq \emptyset$  holds }

```

---

Based on the colored graph  $\mathcal{G}_K$  an upstream token for a concrete flow node is defined in Definition 3 and the implementation is shown in listing 8.12.

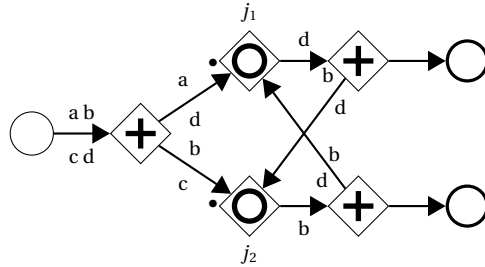
### 8.5.3 Cyclic workflow graphs

In this section we show that the algorithm proposed in section 8.5 works for use cases including cyclic workflow graphs, except some special cases, for which a reasonable semantics is not clear and can be sorted out by static analysis (e.g., Figure 8.1) [60].

The process depicted in Figure 8.1 was colored by rule ColorProcessGraph : N during the deployment of the process and the resulting coloring is indicated by small Latin letters beside the sequence flows. Colors a and b are relevant for the inclusive gateway  $j_1$  and colors c and d are relevant for the inclusive gateway  $j_2$ . Tokens are represented by a dot “•” next to the sequence flow they are contained in.

In the current state depicted in Figure 8.1 we can see that for  $j_1$ , for which a and b are relevant, a directly incoming sequence flow colored with the color a contains a token and therefore all other sequence flows colored with the color a can be ignored.

**Figure 8.1** A symmetric vicious circle [60]



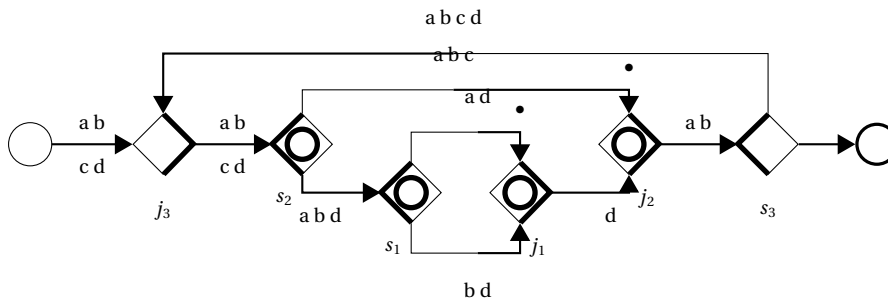
On the other hand, the second directly incoming sequence flow to  $j_1$  colored with the color  $b$  does not contain a token. But there is a sequence flow in the process colored with  $b$  and not  $a$  containing a token which makes such a token an upstream token of the inclusive gateway  $j_1$ . The gateway  $j_1$  has to wait for that token, hence for the activation of the gateway  $j_2$ .

Similarly for the inclusive gateway  $j_2$  and its colors  $c$  and  $d$  where sequence flows colored with the color  $c$  can be ignored for the activation of  $j_2$  as a token is available. Sequence flows containing a token and colored with the color  $d$  and not with the color  $c$  block the activation of  $j_2$ . Hence,  $j_2$  has to wait for the activation of  $j_1$ . This symmetric dependency is considered as a design error with unclear underlying semantics and can be detected using static analysis.

#### 8.5.4 Well structured processes

The coloring of the process for each inclusive or complex gateway simulates the concept of inhibiting and anti-inhibiting paths and so coincide with the *Q-semantics* [60].

**Figure 8.2** A vicious circle in well structured process [60]



In the example of a well structured process depicted in Figure 8.2 the coloring for both merging inclusive gateways ( $j_1, j_2$ ) is shown. Similarly as in Figure 8.1 the colors a and b are relevant for the gateway  $j_1$  and the colors c and d are relevant for the gateway  $j_2$ . It can be observed that the gateway  $j_1$  blocks activation of gateway  $j_2$  but not vice-versa.

### 8.5.5 Non-separable processes

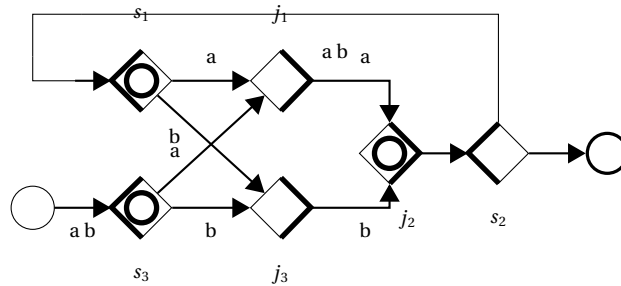
To complete our examples, we also show that the proposed algorithm works correctly on non-separable process workflows. The example shown in Figure 8.3 [60] is showing a process colored with colors a and b relevant to the only inclusive join  $j_2$ . For the sake of brevity we do not color incoming paths of inclusive gateways with only one incoming sequence flow (splits  $s_1$  and  $s_3$ ).

It can be observed that in all of those cases no two inclusive gateways are mutually blocking each other, or themselves.

---

**Figure 8.3** A non-separable processes [60]

---



## 8.6 Transiting to the technical level ground model

In this section the transition of the ground model to the technical level will be completed as described in chapter 3. Function and rule signatures will be further refined. On the highest business level signatures contained event input parameters encoded in their names. Those were transformed using the 4-step transition approach. The result is available in Appendix A. Now we will remove the output parameters from the business level signatures with the following rules:

- if the name starts with a sequence corresponding to its output parameter, or
- if the name ends with a sequence corresponding to its output parameter, and
  - if such an operation will not result in a zero length name, and
  - no two such operations will result in the same name



⇒ it will be cropped.

After the signature name refinement, the analogy between function application and selection [88] will be applied and the result will be grouped using a universe selector. This will result in the following code structure. For the sake of brevity the implementation of derived functions and rules is left out in most of the cases and is included only if some manual refinements during this transformation. In all other cases the implementation of the corresponding derived functions and rules can be studied in Appendix B. The SEQUENCE\_FLOWS related functions are shown in listing 8.13. The

---

**Listing 8.13** Technical level of SEQUENCE\_FLOWS related functions and rules

---

```
1 SEQUENCE_FLOWS {  
2   static source → FLOW_NODES  
3   static target → FLOW_NODES  
4   static gateCondition → EXPRESSIONS  
5 }
```

---

results are consistent with [1, tab. 8.51].

In listing 8.14 the FLOW\_NODES related functions and rules can be seen. It contains the related attributes from [1, tab. 8.52] and all common behavior functions and rules. Here some manual work was added too. The rule `InputBehavior : FLOW_NODES → MULTISSET(INSTANCES)` and rule `OutputBehavior : FLOW_NODES × MULTISSET(INSTANCES)` were refined to show the usage and composition of all the merging and splitting behaviors. We continue with the different flow node type universes starting with the ACTIVITIES as shown in listing 8.15. At this point the ACTIVITIES universe includes only the static `defaultSequenceFlow : ACTIVITIES → SEQUENCE_FLOWS` from [1, tab. 10.3] and the controlled `lifeCycleState : ACTIVITIES × INSTANCES → LIFE_CYCLE_STATES` from [1, tab. 10.4]. This is due to the fact that in this work we do not target to formalize the whole BPMN specification, but rather to show a specific approach how this can be done, incorporating certain clarifications, refinements, and extensions.

Next, the relevant functions and rules for the EVENTS universe are shown in listing 8.16 with the related TRIGGERS universe in listing 8.17. The last flow node type present in the ground model are gateways. The relevant functions and rules grouped for the related GATEWAYS universe are then shown in listing 8.18.

In this chapter we additionally added the CONTEXTS and NOTIFICATIONS universes, which can be seen in listing 8.19 after applying the 4-step transition approach.

---

**Listing 8.14** Technical level of FLOW\_NODES related functions and rules

---

```
1 FLOW_NODES {
2   static parentFlowNode : flowNodes → flowNodes
3   derived incoming → SET(SEQUENCE_FLOWS)
4   derived outgoing → SET(SEQUENCE_FLOWS)
5   derived canPass : SEQUENCE_FLOWS X GATE_BEHAVIOR → BOOLEAN
6   abstract derived firingInstances → MULTISSET(INSTANCES)
7   abstract derived firingTokens → SET(TOKENS)
8
9   rule ActivationBehavior → MULTISSET(INSTANCES)
10  rule Split : INSTANCES
11
12  rule InputBehavior
13  rule InputBehavior =
14    if |self.incoming| = 1 ifthen
15      self.SoleInputBehavior
16    else
17      self.MergeBehavior ∈ { ParallelMergeBehavior, ExclusiveMergeBehavior,
18                             InclusiveMergeBehavior, ComplexMergeBehavior,
19                             MutuallyExclusiveMergeBehavior }
20
21  rule OutputBehavior : MULTISSET(INSTANCES)
22  rule OutputBehavior(flowNode, instances) =
23    ∀ instance ∈ instances
24      if |self.outgoing| = 1 ifthen
25        self.SoleOutputBehavior(instance)
26      else
27        self.SplitBehavior(instance) ∈ { ParallelSplitBehavior,
28                                         ExclusiveSplitBehavior,
29                                         InclusiveSplitBehavior }
30
31  rule Transition
32  rule ColorProcessGraph
33 }
```

---

---

**Listing 8.15** Technical level of ACTIVITIES related functions and rules

---

```
1 ACTIVITIES {
2   static default : SEQUENCE_FLOWS
3   monitored completed : INSTANCES → BOOLEAN
4   monitored interrupted : INSTANCES → BOOLEAN
5   controlled lifeCycleState : INSTANCES → LIFE_CYCLE_STATES
6
7   rule Transition
8 }
```

---

---

**Listing 8.16** Technical level of EVENTS related functions and rules

---

```
1 EVENTS {
2   static triggers → SET(TRIGGERS)
3   controlled eventOccured : INSTANCES → BOOLEAN
4   derived triggerName → STRINGS
5
6   rule Behavior
7   rule Transition
8 }
9
10 rule CATCH_EVENTS {
11   static parallelMultiple → BOOLEAN
12
13   rule Transition
14   rule StartEventBehavior
15 }
16
17 rule THROW_EVENTS.Transition
18
19 rule START_EVENTS.Transition
20 rule INTERMEDIATE_EVENTS.Transition
21 rule END_EVENTS.Transition
22
23 rule INTERMEDIATE_CATCH_EVENTS.Transition
24 rule INTERMEDIATE_THROW_EVENTS.Transition
25
26 BOUNDARY_EVENTS {
27   static attachedTo → ACTIVITIES
28   rule Transition
29 }
30
31 rule SUB_PROCESS_EVENTS.Transition
32 rule TOP_LEVEL_EVENTS.Transition
```

---

---

**Listing 8.17** Technical level of TRIGGERS related functions and rules

---

```
1 TRIGGERS {
2   derived triggerName → STRINGS
3   rule Throw : INSTANCES
4 }
5
6 static CONDITIONAL_TRIGGERS.conditionExpression → EXPRESSIONS
7 rule EXCEPTION_TRIGGERS.Throw : INSTANCES
```

---

---

**Listing 8.18** Technical level of GATEWAYS related functions and rules

---

```
1 GATEWAYS {
2   static direction → GATEWAY_DIRECTION
3   rule Transition
4 }
5
6 rule DATA_BASED_GATEWAYS.Transition
7 rule PARALLEL_GATEWAYS.Transition
8 rule EXCLUSIVE_GATEWAYS.Transition
9 rule INCLUSIVE_GATEWAYS.Transition
10
11 COMPLEX_GATEWAYS {
12   static activationCondition → EXPRESSION
13   controlled ignoreDuringReset → SET(SEQUENCE_FLOWS)
14   controlled waitingForStart → BOOLEAN
15   rule Transition
16 }
```

---

---

**Listing 8.19** Technical level of CONTEXTS and NOTIFICATIONS related functions and rules

---

```
1 // Auxiliary functions
2 monitored notifications → PRIORITY_QUEUE
3
4 CONTEXTS {
5   derived waitingTasks → SET(TASKS)
6   monitored instance → INSTANCES
7   monitored parent → CONTEXTS
8 }
9
10 NOTIFICATIONS {
11   shared context → CONTEXTS
12   shared flowNode → FLOW_NODES
13   monitored occurrenceTime → TIME
14 }
15
16 EXCEPTION_NOTIFICATIONS {
17   controlled code → STRINGS
18   controlled name → STRINGS
19
20   rule PropagateNotification
21 }
22
23 SIGNAL_NOTIFICATIONS {
24   controlled name → STRINGS
25   rule PublishNotification
26 }
27
28 shared MESSAGE_NOTIFICATIONS.payload → STRINGS
```

---

*A conclusion is the place where you got tired thinking.*

— Martin H. Fischer

## Chapter 9

# Conclusion

### 9.1 Results

In this work we evaluated workflow techniques, especially the BPMN specification and its underlying meta-model in [68, 1]. We identified some pitfalls in the existing specification and proposed corrections and improvements. Although the BPMN specification is claiming a formal underlying semantics some ambiguities and inconsistencies still remain and were already addressed by others using different formalisms. We evaluated the major approaches in literature relevant to this thesis. We believe that using a unified, unambiguous and clear modeling notation along with formal semantics will lead to a steeper learning curve and greater understanding of processes in general and in particular processes inside a company.

In chapter 6 we showed an approach of a behavioral decomposition of the BPMN 2.0 control flow concept. Regarding gateways in section 4.4 we introduced an improved gateway class hierarchy which preserves the semantics in [1] and further allows us to reuse existing behavior definitions and even extend the BPMN 2.0 standard by some other gateway stereotypes. Since we started with the merge behavior of incoming and split behavior of outgoing sequence flows. The split and merge behaviors, which are shared across other flow node types, such as activities or events, can be simply reused in their transition rules as discussed in section 6.5. The different flow node types can then be defined by composing different behavior patterns as shown for activities in Figure 1.5. The goal of this decomposition is to define a behavior once in a way that it is reusable across the defined system. This enables that such a model is better graspable and understandable due to, e.g., smaller size of the decomposition blocks. Also such a resulting ground model is easily extendable with other behaviors. This was shown on a theoretical extension example in section 6.6, where the discussed ground model was extended by a sole exclusive gateway type. We leave the discussion about the correct behavior of exclusive gateways to some future work.

Regarding events we merged the two attributes, `StartEvent#isInterrupting:boolean` [1, tab. 10.87] and `BoundaryEvent#cancelActivity:boolean` [1, tab. 10.91], marking the common behavior of the corresponding classes, into *Interrupting*. This class allows us to define the behavior, actually interrupting an instance, in one place and reuse it in both classes, `SubProcessEvent` and `BoundaryEvent`, classes. Additionally the cancel trigger is expected to have that same *Interrupting* behavior and can now also reuse it. Furthermore we defined the common behavior for intermediate events by inserting the *IntermediateEvent* class into the meta-model. The classification of events into catch events and throw events (see group *A* in Figure 4.10) defines the event part of the `WorkflowTransition`. The classification of events into start events, end events, intermediate events and boundary events (see group *B* in Figure 4.10) defines the control flow part of the `WorkflowTransition` [96]. The special case of `ImplicitThrowEvent` is then defined on a lower level of the ground model in Part III.

Thus, the new model has a more concise structure, avoiding duplication, which can otherwise lead to severe inconsistencies in WFI implementations. During our refinements we focused on compatibility with the original meta-model. E.g., the removal of `StartEvent#isInterrupting:boolean` and `BoundaryEvent#cancelActivity:boolean` attribute can be simulated by checking if the concrete flow node is an instance of *Interrupting* class. In this way a removable compatibility layer can be provided. In chapter 6 we provided the formal semantics for most of the classes in the new meta-model shown in Figure 4.10. The missing `ImplicitThrowEvent` is then defined on lower abstraction level in Part III as `ThrowImplicitNotifications`. The `TopLevelEvent` and `SubProcessEvent` classes will come into focus as soon as the different trigger types are defined. The *Interrupting* class should perform additional actions to cancel the encompassing or attached activity. This topic is related to instances and instantiation of activities and their life-cycle. Both of the undefined areas are out of the scope of this thesis.

Based on the ASM method, abstraction levels were introduced to support the model refinement during the development process of the ground model. Two basic abstraction levels were defined in chapter 3. Their different structures and purposes were outlined and the transition between them was also presented in chapter 3. The transition from the business level document to a technical level document was described in section 3.3. Using conventions and transition rules defined in section 3.3 the complete transition of the ground model on the business level defined in chapter 4, 5 and 6 to technical level in 7 and with later vertical refinements in the direction of a WFI in 8 is shown in Appendix A.

The 4-step transition approach, illustrated in this thesis, shows one possible way how to deal with refinements. The transition rule framework should make the transition machine configurable in such a way that it can be used for different system designs by adjusting the rules and conventions to satisfy the concrete needs. The results will be used to develop further ideas concerning tool support for working with the ASM model for system specification. The ASM models are described in text documents, as no adequate tool support is available for now. Thus keeping the models consistent throughout the refinement process or when handling change requests later on, is challenging. Being able to unambiguously and clearly pass the informa-

tion of what is going to be implemented in order to match the requirements is often not trivial. We proposed conventions for describing functions, rules and associated concepts as well as a stepwise refinement process based on transition rules. The conventions and rules are also used vice-versa as the basis for consistency checks, which are needed to guarantee that technical level models are consistent with business level ones.

The business level with its refinement steps is intended to be the communication basis between the product owner and the product designer. Similarly for the technical level, which is dedicated to the product designer and the product implementer. For now this approach has only been tested on some small examples, e.g., [97] or within this thesis, but expect to be promising. It will soon be tested on our largest ongoing project using the ASM method to formalize the entire BPMN 2.0 specification.

During the writing of this thesis an ASM Ground Model  $\text{\LaTeX}$  package was developed (see Appendix C), which was used in relevant parts of this thesis to support consistency during migration, definition and refinements of the present constructs.

In Part III the new refined meta-model from chapter 4 and 5 is matched to the notification concept and wraps it in a specification of the modeling notation with a vertical refinement targeting a WFI. The new decomposition and related refinements presented are part of our ongoing project, where we are working on formally describing and improving the BPMN meta-model to eliminate ambiguities and to allow a clear design further vertically refined in the direction of the WFI as a common ground of consistent WFI implementations in future products. The notification concept, shown in this paper, allows the WFI to control and observe the processes deployed to it. It implements the communication concept using messages or signals with other processes. It also enables the WFI to instantiate new processes and to react to some events, e.g. unhandled error, escalation or terminate trigger events. The context concept, defined in section 8.1, is used as a communication medium to forward the notifications to different parts of running processes and activities in the WFI. The notifications are forwarded using the event forwarding concepts [1], formally defined in section 8.3, 8.3.2 and 8.3.3. The link events were left out since they are used only to break sequence flows to solve some graphical presentation limitations in the same process level [1, sec. 10.4.5] and this can be handled by the rule `WorkflowTransition : FlowNodes` [15].

In section 8.3.1 we have shown the creation of notifications for incoming messages and signals using pools. One can see the similarity between a message and a signal. The two apparent differences are that a message specifies a concrete target (i.e. catching event) and carries a more complex payload [1, fig. 8.30, tab. 8.48], while a signal can be caught by any, and possibly more than one, catching event and does not carry any additional payload [1, fig. 10.93]. Since we cannot model a message flow across different processes possibly running in different WFEs we match the message target in section 8.3.2 using the name of the message. Another possibility is to define `messageFlowNameOfMessage` and match the target flow node by that parameter, since a message flow also specifies a name [1, fig. 8.30]. We chose to match the target by the name of the message and not message flow to demonstrate the similarity between messages and signals and because message flow is a part of

collaboration [1, sec. 9] which is not necessary to claim neither *Process Execution Conformance* [1, sec. 2.2] nor *BPEL Process Execution Conformance* [1, sec. 2.3]. The concepts presented in this thesis, the context concept and the notification concept, are the core concepts currently used for our work in progress abstract specification of a WFE.

On the technical level we further refined the activation concept used in inclusive and complex gateways. The core of this part is the modified Dijkstra's algorithm [59] represented by the rule `ColorProcessGraph : N` described in section 8.5. This coloring may be computed during deployment of a process into a WFE for every flow node, which may need it and so speedup the execution of instances of such a process. The computation complexity played a role while choosing Dijkstra's algorithm as a base for the coloring algorithm, as its asymptotic complexity which be optimized up to  $\mathcal{O}(|E| + |N|\log|N|)$  [98], where  $|E|$  is the number of relevant sequence flows and  $|N|$  the number of relevant flow nodes for the computation of the upstream token. In the ASM pseudo code representation shown in listing 8.11 we did not further focus on execution optimization and left this open for a concrete implementation in some future work. Emphasis was placed on brevity, simplicity and reusability of the algorithm. A reference implementation of the coloring algorithm was done in Ruby [91] as a proof of concept. Herewith have presented a possible refinement of the non-primitive activation concept of inclusive and complex gateways in the BPMN ground model [15, 17]. We demonstrated correct work of the algorithm and also identified cases, where the proposed algorithm will not work, such as *Symmetric Vicious Circles* [60]. We agree that such cases may be considered as design errors where the underlying semantics is unclear and may be ruled out during development or deployment of a process by static analysis [99].

To run a process using our ground model a ASM representation of such process is necessary. Both the graphical representation of the BPMN process and the ASM code should hold the semantics of the process and therefore they should be convertible in both directions. A graphical representation of a process should be easily converted into a formal yet intelligible ASM code. At this level the user can define more detailed information about the process, constraints, and dependencies needed to deterministically run the code. This additional information is not present in the BPMN graphical representation and will therefore not be added into a BPMN diagram when converting it from the corresponding ASM code. This is true also for the absolute position of the flow elements in a BPMN diagram, which is not present in the ASM code. But any change in the semantics of the model should be reflected in both representations if a conversion is performed either way, the graphical representation and an easy to read formal description of the process. Such a cleartext description should be still easy enough to be understood by a wide audience but still clear, unambiguous and formal. The process of converting the graphical representation to the formal description (code) has to be a deterministic, easy, and as simple as possible routine job. Every conversion of a BPMN diagram should result in the exactly same code. The clear and unambiguous graphical representation of a business process should be easily convertible to a formal, easy to read code, which can be converted back to the same graphical representation without any loss of information or information change in the model semantics. The detailed description



of a flow node using ASMs can be converted back to a BPMN diagram without the detailed description, which is not supported by BPMN. But still the user will be able to update the graphical representation when the code changes.

Thus, many questions are still open. We will continue our work with a detailed review of the requirements identified so far, integrating the latest trends in this area, before going into more details concerning our basic model.

## 9.2 Future Work

A couple of ideas for future work were raised during the work on this thesis. We are planning to use this 4-step transition approach in our future work using ASMs for formal specification and additional proof the concept by studying different writing techniques on different levels of abstraction, (semi-)automatic transition from the business level to the technical level of abstraction and preserving consistency between the involved documents. We expect this to contribute solving problems we are currently facing when changing a complex formal specification of a system, which now requires enormous manual effort and increases the risk of introducing inconsistencies or typographical errors. Also a definition of a coherent transition machine resulting in proper tool support should be done in order to complete this idea.

Furthermore the ground model of the BPMN specification need to be completed. This is part of our currently ongoing project were topics, e.g., instantiation, link intermediate events, event-based gateways, tasks and other parts of the BPMN specification not included in the ground model presented in this thesis are focused. This comes hand in hand with the refinements targeting a WFI. The first step is to agree on a Target Architecture (TA), e.g., the interfaces needed for a complete WFE. Those need to be defined and refined so the step between such ground model and an implementation is more or less trivial. This includes message and signal matching, further refinements of the notification concept, fixing the context interface and optimizing used algorithms.

Furthermore, since the conversion process between BPMN diagrams and ASM code should be deterministic, easy, and simple routine job (except from purely graphical information), such conversions should be automated in the future. The universe aliases shown listing 7.1 are those explicitly used in our ground model. But a more general approach can be chosen. Since we already defined used universes using the meta-model and also in chapter 5 a automated approach should be considered.

Hence there is still much work to be done and a lot open issues, some of which we try to address in other related projects.



# Bibliography

- [1] Object Management Group (OMG): Business Process Model and Notation (BPMN) 2.0. [www.omg.org/spec/BPMN/2.0/](http://www.omg.org/spec/BPMN/2.0/). Accessed: 2011-08-28. (August 2011)
- [2] Börger, E., Stärk, R.F.: Abstract State Machines - A Method for High-Level System Design and Analysis. Springer-Verlag (2003)
- [3] Börger, E., Schulte, W.: A Programmer Friendly Modular Definition of the Semantics of Java. In: Formal Syntax and Semantics of Java, London, UK, UK, Springer-Verlag (1999) 353–404
- [4] Börger, E., Schulte, W.: Modular Design for the Java Virtual Machine Architecture. In Börger, E., ed.: Architecture Design and Validation Methods. Springer Berlin Heidelberg (1999) 297–357
- [5] Kutter, P.W., Pierantonio, A.: The Formal Specification of Oberon. *Journal of Universal Computer Science* **3**(5) (May 1997) 443–503
- [6] Object Management Group (OMG): Business Process Model and Notation (BPMN) 2.0.1. [www.omg.org/spec/BPMN/2.0.1/](http://www.omg.org/spec/BPMN/2.0.1/). Accessed 2011-08-20. (August 2013)
- [7] ISO/IEC 19510:2013: Information technology – Object Management Group Business Process Model and Notation. First edition edn. Number 19510 in International Standard. ISO, Geneva, Switzerland, ISO copyright office, Case postale 56, CH-1211 Geneva 20, Switzerland (September 2013)
- [8] ter Hofstede, A.H.M., van der Aalst, W.M., Adams, M., Russell, N., eds.: Modern Business Process Automation - YAWL and its Support Environment. Springer (2010)
- [9] Fleischmann, A., Schmidt, W., Stary, C., Obermeier, S., Börger, E.: Subject-Oriented Business Process Management. Springer, Berlin (2012)
- [10] Reichert, M., Weber, B.: Enabling Flexibility in Process-Aware Information Systems: Challenges, Methods, Technologies. Springer, Berlin (2012)
- [11] van der Aalst, W.M.: The Application of Petri Nets to Workflow Management. *Journal of Circuits, Systems, and Computers* **8**(1) (1998) 21–66

- [12] van der Aalst, W.M.: Workflow Verification: Finding Control-Flow Errors Using Petri-Net-Based Techniques. In van der Aalst, W.M., Desel, J., Oberweis, A., eds.: Business Process Management. Volume 1806. Springer Berlin / Heidelberg (2000) 19–128
- [13] van der Aalst, W.M.: Challenges in Business Process Management: Verification of Business Processing Using Petri Nets. *Bulletin of the EATCS* **80** (June 2003) 174–199
- [14] Börger, E.: A Complete Precise Interpreter Model for S-BPM. (2011)
- [15] Börger, E., Thalheim, B.: A method for verifiable and validatable business process modeling. *Advances in Software Engineering, LNCS* **5316** (2008) 59–115
- [16] Börger, E., Thalheim, B.: Modeling Workflows, Interaction Patterns, Web Services and Business Processes: The ASM-Based Approach. In: *Proceedings of the 1st international conference on Abstract State Machines, B and Z*, Springer Berlin / Heidelberg (2008) 24–38
- [17] Börger, E., Sörensen, O.: BPMN Core Modeling Concepts: Inheritance-Based Execution Semantics. In Embley, D.W., Thalheim, B., eds.: *Handbook of Conceptual Modeling: Theory, Practice and Research Challenges*. Springer-Verlag (2011)
- [18] Fleischmann, A., Schmidt, W., Stary, C., Obermeier, S., Börger, E.: A Precise Description of the S-BPM Modeling Method. In: *Subject-Oriented Business Process Management*. Springer, Berlin (2012)
- [19] Wong, P.Y., Gibbons, J.: A Process Semantics for BPMN. In Liu, S., Maibaum, T., Araki, K., eds.: *Formal Methods and Software Engineering*. Volume 5256 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg (2007) 355–374
- [20] Dijkman, R.M., Dumas, M., Ouyang, C.: Formal Semantics and Analysis of BPMN Process Models using Petri Nets (2007)
- [21] Dijkman, R.M., Dumas, M., Ouyang, C.: Formal Semantics and Automated Analysis of BPMN process models. Technical report, Queensland University of Technology, Faculty of Science and Technology (2007)
- [22] Dijkman, R.M., Dumas, M., Ouyang, C.: Semantics and Analysis of Business Process Models in BPMN. *Inf. Softw. Technol.* **50**(12) (November 2008) 1281–1294
- [23] van der Aalst, W.M., ter Hofstede, A.H.M.: YAWL: Yet Another Workflow Language. *Information Systems* **30** (2003) 245–275

- [24] Koniewski, R., Dzielinski, A., Amborski, K.: Use of Petri Nets and Business Processes Management Notation in Modelling and Simulation of Multimodal Logistics Chains. In Borutzky, W., Orsoni, A., Zobel, R., eds.: 20th European Conference on Modelling and Simulation, ECMS (2006) 4
- [25] Object Management Group (OMG): Business Process Modeling Notation 1.0. [www.bpmn.org/Documents/OMG\\_Final\\_Adopted\\_BPMN\\_1-0\\_Spec\\_06-02-01.pdf](http://www.bpmn.org/Documents/OMG_Final_Adopted_BPMN_1-0_Spec_06-02-01.pdf) (February 2006) OMG Final Adopted Specification.
- [26] Alfresco Software, Inc: Activiti BPM Platform. <http://activiti.org/>. Accessed: 2014-02-23.
- [27] Bonitasoft: Bonita BPM. <http://www.bonitasoft.com>. Accessed 2014-02-23.
- [28] Together Teamsolutions Co., Ltd.: Together XPDL and BPMN Workflow Server. <http://shark.enhydra.org>. Accessed 2014-02-23. (2011)
- [29] Red Hat, Inc.: JBoss Enterprise SOA Platform 5 - JBPM Reference Guide. Red Hat, Inc. 5.3.0 edn. (2012) Accessed: 2012-10-12.
- [30] Mettraux, J., Kalmer, K., Meyers, R., de Mik, H.C., Kohlbecker, A., Barnaba, M., Neskovic, G., Stults, N., Pudeyev, O., Gfeller, M., Brindisi, P., Boettcher, B., Bryant, D., Pospíšil, J.: Ruote - a Ruby Workflow Engine. <https://github.com/jmettraux/ruote>. Accessed 2014-06-05.
- [31] Dumas, M., Hofstede, A.H.M.t.: Uml activity diagrams as a workflow specification language. In: Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools. &#171;UML&#187; '01, London, UK, UK, Springer-Verlag (2001) 76–90
- [32] Peterson, J.L.: Petri nets. *ACM Comput. Surv.* **9**(3) (September 1977) 223–252
- [33] Freund, J., Rücker, B., Heinninger, T.: *Praxishandbuch BPMN Incl. BPMN 2.0*. Carl Hanser Verlag München Wien (2010)
- [34] Weske, M., Decker, G., Grosskopf, A., Wagner-Boysen, S.: BPMN - Business Process Modeling Notation. poster
- [35] Berlin, B.O.: BPMN 2.0 - Business Process Model and Notation. poster (2011)
- [36] Polančič, G., Rozman, T.: Business Process Modelling Notation (BPMN) Poster. poster (October 2008)
- [37] Camunda Services GmbH: BPMN.info Website. [www.bpmn.info](http://www.bpmn.info). Accessed: 2014-12-03. (March 2014)
- [38] Gurevich, Y.: A new thesis (abstract). *American Mathematical Society* **6**(4) (1985) 317

- [39] Gurevich, Y.: Evolving Algebras 1993: Lipari Guide. In Börger, E., ed.: *Specification and Validation Methods*. Oxford University Press (1995) 9–36
- [40] Gurevich, Y.: Sequential abstract state machines capture sequential algorithms. *ACM Transactions on Computational Logic* **1**(1) (2000) 77–111
- [41] Blass, A., Gurevich, Y., Bussche, J.D.: Abstract State Machines and Computationally Complete Query Languages. In *Inf. Comput.* [100] 22–33
- [42] Blass, A., Gurevich, Y.: Abstract state machines capture parallel algorithms. *ACM Transactions on Computational Logic* **4**(4) (2003) 578–651
- [43] Blass, A., Gurevich, Y.: Abstract state machines capture parallel algorithms: Correction and extension. *ACM Transactions on Computational Logic* **9**(3) (2008) 1–32
- [44] Kossak, F., Illibauer, C., Geist, V., Kubovy, J., Natschläger, C., Ziebmayer, T., Kopetzky, T., Freudenthaler, B., Schewe, K.D.: *A Rigorous Semantics for BPMN 2.0 Process Diagrams*. Springer (2014)
- [45] Natschläger, C.: Deontic BPMN. In Hameurlain, A., Liddle, S., Schewe, K.D., Zhou, X., eds.: *Database and Expert Systems Applications*. Volume 6861 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg (2011) 264–278
- [46] Natschläger, C., Kossak, F., Schewe, K.D.: Deontic BPMN: a powerful extension of BPMN with a trusted model transformation. *Software & Systems Modeling* (March 2013) 1–29
- [47] Abrial, J.R.: *The B-book - assigning programs to meanings*. pCambridgeUniversityPress, New York and NY and USA (1996)
- [48] Abrial, J.R., Lee, M., Neilson, D., Scharbach, P., Sørensen, I.: The B-method. In Prehn, S., Toetenel, H., eds.: *VDM '91 Formal Software Development Methods*. Volume 552 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg (1991) 398–405
- [49] Systems, D., at the University of Southampton, S.E.R.G.: Event-B and the Rodin Platform. <http://www.event-b.org/>. Accessed: 2014-01-30.
- [50] CoreASM.org: CoreASM Website. <http://www.coreasm.org>. Accessed: 2014-01-30.
- [51] Farahbod, R., Gervasi, V., Glässer, U.: CoreASM: An Extensible ASM Execution Engine. In: *Abstract State Machines*. Volume 77. (2007) 77–103
- [52] Farahbod, R., Gervasi, V.: *Design and Specification of the CoreASM Execution Engine*. Technical report, Computing Science, Simon Fraser University, Burnaby, B.C., Canada (2005)

- [53] Stärk, R.F., Schmid, J., Börger, E.: Java and the Java Virtual Machine. Definition, Verification, Validation. Springer-Verlag (2001)
- [54] Kossak, F., Illibauer, C., Geist, V.: Modelling the Semantics of BPMN Models as an ASM Ground Model. Software Competence Center Hagenberg (2013)
- [55] Thomas, Z.: Zielarchitektur VMI-HLM - Actual status presentation. unpublished (2012) Software Competence Center Hagenberg.
- [56] Geist, V.: Integrated Executable Business Process and Dialogue Specification. PhD thesis, Johannes Kepler University Linz, Alternberger Strasse 69, 4040 Linz, Austria (June 2011)
- [57] Hayes, P.J.: The Frame Problem and Related Problems in Artificial Intelligence. Technical report, Computer Science Department, Stanford University (November 1971)
- [58] Kossak, F., Illibauer, C., Geist, V.: Event-based gateways: Open questions and inconsistencies. In Mendling, J., Weidlich, M., eds.: Business Process Model and Notation. Volume 125 of Lecture Notes in Business Information Processing. Springer Berlin Heidelberg (2012) 53–67
- [59] Dijkstra, E.W.: A Note on Two Problems in Connexion with Graphs. In: Numerische Mathematik. Volume 1. Springer-Verlag (December 1959) 269–271
- [60] Völzer, H.: A new semantics for the inclusive converging gateway in safe processes. In Hull, R., Mendling, J., Tai, S., eds.: Proceedings of the 8th international conference on Business process management. Volume 6336 of BPM'10., Berlin, Heidelberg, Springer Berlin Heidelberg (2010) 294–309
- [61] vom Brocke, J., Rosemann, M., eds.: Handbook on Business Process Management 1: Introduction, Methods, and Information Systems. 1st edn. Springer Publishing Company, Incorporated (2010)
- [62] Levy, F.: How Technology Changes Demands for Human Skills. OECD Education Working Papers (45) (March 2010) 19
- [63] Davenport, T.H.: Thinking for a Living: How to Get Better Performance and Results from Knowledge Workers. illustrated edition edn. Harvard Business School Press, Boston, Mass. (2005)
- [64] Dadam, P., Reichert, M., Rinderle-Ma, S.: Prozessmanagementsysteme: Nur ein wenig Flexibilität wird nicht reichen. Informatik-Spektrum **34**(4) (August 2011) 364–376
- [65] Software AG: Modeling BPMN 2.0 in ARIS. Software AG, Uhlandstrasse 12D-64297, Darmstadt, Germany. Version 9.5 sr edn. (January 2014)

- [66] Terceros, C.A., Leslie, S.: Oracle Fusion Middleware Modeling and Implementation Guide for Oracle BusinessProcess Management. Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065. 11grelease 1 (11.1.1.6.3) edn. (July 2012) Describes how to design and implement business processesusing Oracle Business Process Studio.
- [67] Object Management Group (OMG): Business Process Model and Notation (BPMN) 1.1. [www.omg.org/spec/BPMN/1.1/](http://www.omg.org/spec/BPMN/1.1/) (January 2008)
- [68] Object Management Group (OMG): Business Process Model and Notation (BPMN) 1.2. [www.omg.org/spec/BPMN/1.2/](http://www.omg.org/spec/BPMN/1.2/) (January 2009)
- [69] Object Management Group (OMG): OMG Unified Modeling Language (OMG UML) 2.2. <http://www.omg.org/spec/UML/2.2>. Accessed 2012-07-05 (2009)
- [70] Organization for the Advancement of Structured Information Standards (OASIS): Web Services Business Process Execution Language Version 2.0. [https://www.oasis-open.org/standards](http://www.oasis-open.org/standards) (April 2007) OASIS Standard.
- [71] Object Management Group (OMG): Case Management Model and Notation (CMMN). <http://www.omg.org/spec/CMMN/1.0/Beta1/PDF/> (January 2013) FTF Beta 1.
- [72] Clarke, E.M., Wing, J.M., Alur, R., Cleaveland, R., Dill, D., Emerson, A., Garland, S., German, S., Guttag, J., Hall, A., Henzinger, T., Holzmann, G., Jones, C., Kurshan, R., Leveson, N., McMillan, K., Moore, J., Peled, D., Pnueli, A., Rushby, J., Shankar, N., Sifakis, J., Sistla, P., Steffen, B., Wolper, P., Woodcock, J., Zave, P.: Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys* **28**(4) (1996)
- [73] Börger, E., Rosenzweig, D.: A mathematical definition of full Prolog. *Science of Computer Programming* **24** (1995) 249–286
- [74] Börger, E., Cavarra, A., Riccobene, E.: An asm semantics for uml activity diagrams. In Rus, T., ed.: *Algebraic Methodology and Software Technology*. Volume 1816 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg (2000) 293–308
- [75] Barnett, M., Börger, E., Gurevich, Y., Schulte, W., Veanes, M.: Using Abstract State Machines at Microsoft: A Case Study. [100] 367–379
- [76] Kubovy, J., Geist, V., Kossak, E.: A Formal Description of the ITIL Change Management Process Using Abstract State Machines. 2012 23rd International Workshop on Database and Expert Systems Applications **0** (September 2012) 65–69
- [77] Kubovy, J., Auer, D., Küng, J.: Behavior-Based Decomposition of BPMN 2.0 Control Flow. In: *ICEIS 2014 - Proceedings of the 16th International Conference on Enterprise Information Systems*, Lisabon, Portugal, Institute for



Systems and Technologies of Information, Control and Communication (IN-STICC), SciTePress (April 2014) 9

- [78] Kossak, F., Illibauer, C., Geist, V., Natschläger, C., Kopetzky, T., Ziebermayr, T., Trenker, T., Kubovy, J., Freudenthaler, B., Schewe, K.D.: Vertical Model Integration (VMI). Project Meetings (2012)
- [79] Jech, T.: Set theory. Springer Monographs in Mathematics. Springer Berlin Heidelberg (2003)
- [80] Lamport, L., Paulson, L.C.: Should your specification language be typed. *ACM Trans. Program. Lang. Syst.* **21**(3) (May 1999) 502–526
- [81] Wildberger, N.J.: A new look at multisets. preprint (2003)
- [82] Castillo, G., Gurevich, Y., Stroetmann, K.: Typed Abstract State Machines. Unpublished (1998) 1–25
- [83] Gurevich, Y.: Evolving Algebras: An Attempt To Discover Semantics (1993)
- [84] Office of Government Commerce (OGC): ITIL V3 Service Transition. The Stationery Office (2007)
- [85] Pressman, R.S.: Software Engineering - A Practitioner's Approach. Thomas Casson (2001)
- [86] van Gorp, P., Dijkman, R.: A Visual Token-based Formalization of BPMN 2.0 based on In-place Transformations. *Information and Software Technology* **2**(55) (2012) 365–394
- [87] Office of Government Commerce (OGC): ITIL Version 3: Service Operation. The Stationery Office (2007)
- [88] Odersky, M.: Objects + Views = Components? In Gurevich, Y., Kutter, P.W., Odersky, M., Thiele, L., eds.: *Abstract State Machines 2000*. Volume 1912 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg (March 2000) 50–68
- [89] Mogensen, T.Æ.: Basics of Compiler Design. Anniversary edition edn. lulu.com, Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, Denmark (August 2010)
- [90] Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification. 3 edn. Addison-Wesley Longman, Upper Saddle River, NJ (2005)
- [91] Flanagan, D., Matsumoto, Y.: The Ruby Programming Language. O'Reilly, 1005 Gravenstein Highway North, Sebastopol, CA 95472 (January 2008)
- [92] Nørmark, K.: Object-oriented Programming in C# for C and Java programmers. Department of Computer Science, Aalborg University, Denmark (2010)

- [93] Kubovy, J., Küng, J.: Formal description of Control Flows in Business Process Model and Notation Workflow Interpreter using Abstract State Machines. Fourteenth International Conference on Computer Aided Systems Theory (February 2013)
- [94] Alfresco Software, Inc: Activiti 5.10 User Guide. <http://activiti.org/userguide/index.html>. Accessed 2012-10-08. (2012)
- [95] Castillo, G., Winter, K.: Model Checking Support for the ASM High-Level Language. In Graf, S., Schwartzbach, M., eds.: Tools and Algorithms for the Construction and Analysis of Systems. Volume 1785 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2000) 331–346
- [96] Börger, E.: Modeling Workflow Patterns from First Principles. In Parent, C., Schewe, K.D., Storey, V., Thalheim, B., eds.: Conceptual Modeling - ER 2007. Volume 4801 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2007) 1–20
- [97] Kubovy, J., Rady, M., Auer, D., Küng, J.: Transition between different abstraction levels in an Abstract State Machine (ASM) Ground Model. 2013 24rd International Workshop on Database and Expert Systems Applications (August 2013) 227–230
- [98] Fredman, M.L., Tarjan, R.E.: Fibonacci Heaps And Their Uses In Improved Network Optimization Algorithms. *J. ACM* **34**(3) (July 1987) 596–615
- [99] Dumas, M., Grosskopf, A., Hettel, T., Wynn, M.: Semantics of standard process models with OR-joins. In: Proceedings of the 2007 OTM Confederated international conference on On the move to meaningful internet systems: CoopIS, DOA, ODBASE, GADA, and IS - Volume Part I. OTM'07, Berlin, Heidelberg, Springer-Verlag (2007) 41–58
- [100] Gurevich, Y., Kutter, P.W., Odersky, M., Thiele, L., eds.: Abstract State Machines, Theory and Applications, International Workshop, ASM 2000, Monte Verità, Switzerland, March 19-24, 2000, Proceedings. In Gurevich, Y., Kutter, P.W., Odersky, M., Thiele, L., eds.: Abstract State Machines. Volume 1912 of Lecture Notes in Computer Science., Springer (2000)
- [101] Gurevich, Y.: Evolving Algebras. A Tutorial Introduction. *Bulletin of EATCS* **43** (1991) 264–284
- [102] Gurevich, Y.: Evolving algebras 1993: Lipari guide. In Börger, E., ed.: Specification and validation methods. Oxford University Press, Inc., New York, NY, USA (1995) 9–36
- [103] Drucker, P.F.: Knowledge-worker productivity: The biggest challenge. *California Management Review* **41**(2) (1999) 79–94

- [104] Collins-Sussman, B., Fitzpatrick, B.W., Pilato, M.C.: Version Control with Subversion. O'Reilly & Associates, Inc., Sebastopol, CA, USA (March 2004) For Subversion 1.6.
- [105] Loeliger, J.: Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development. 1 edn. O'Reilly (June 2009)
- [106] Kramer, D.: API Documentation from Source Code Comments: A Case Study of Javadoc. In Johnson-Eilola, J., Selber, S.A., eds.: Proceedings of the 17th Annual International Conference on Computer Documentation. SIGDOC '99, New York, NY, USA, Association for Computing Machinery (ACM) (1999) 147–153



# Index

- Activity, 11, 12, 26, 35–39, 42, 43, 65, 66, 70, 77, 94–96, 98, 102, 108, 113, 114, 118, 132, 154–157, 163, 165, 176, 191–194, 199, 209
- Activity Life-Cycle State, 39, 95, 163, 164, 191
- Actor, 7, 39, 167
- Alternative Path, 38, 121, 179
- Anti-Inhibiting Path, 5, 121, 124, 167, 179
- Environment, 19, 154, 157
- ASM Method, xix, 16
- ASM ground model  $\text{WfX}$  package, 56, 78, 133, 213, 215, 218, 219, 223, 226, 229
- Association, 156
- Basic functions, 16, 20, 161, 215
- Boundary Event, 43, 65, 77, 98, 104, 156, 163, 194, 201
- BPEL Process Execution Conformance, 9, 10, 117
- BPMN<sup>+</sup>, 6
- Business level of ASM abstraction, xix, 15, 18, 19, 23, 24, 26–30, 55, 65, 78–80, 113, 114, 126, 132, 133, 135, 169, 215
- Call Activity, 113, 160
- Cancel Triggers, 132
- Case, 155
- Catch Event, 46, 63, 65, 66, 98, 100, 109, 114, 163, 194, 197, 205
- choose, 21
- Choreography, 9
- Collaboration, 9, 117, 134, 162
- Compensation Triggers, 43
- Complex Gateway, 5, 36, 63, 75, 87, 105, 107, 121, 122, 125, 157, 177, 179, 185, 202, 204
- Conditional Sequence Flow, 35, 38, 49, 62, 64, 70, 108, 110
- Conditional Trigger, 48, 67, 116, 158, 208
- Constraint, 62, 89, 187
- Context, 114–116, 118, 133–135, 205, 208, 209
- Context Tree, 113, 158, 161, 164
- Control Flow, xix, 5, 34, 37, 40, 44, 49, 51–53, 69, 70, 77, 89, 111, 112, 131, 132, 188
- Controlled functions, 81, 95, 98, 105, 191, 192, 195, 202
- Controlled Sequence Flow, 36, 37
- CoreASM Engine, 4, 16
- Data Input, 40
- Data-Based Gateway, 52
- Declaration Signature, 215, 217, 219, 220, 224
- Default Expression, 64
- Default Sequence Flow, 36, 64, 108
- Definition Signature, 215, 217, 221
- Deployment Manager, 120
- Derived functions, 20, 21, 81, 83, 98, 99, 105, 154, 175–177, 180, 189, 192, 195, 202, 215, 221
- Enabling Token, 81
- End Event, 45–47, 52, 78, 102, 155, 199
- Error, 47, 48, 62, 118, 133
- Error Notification, 67, 118, 204, 205

Error Trigger, 47, 67, 68, 109, 118  
 Escalation, 47, 62, 118, 133  
 Escalation Notification, 204, 205  
 Escalation Trigger, 47, 67, 68, 109, 198  
 Event, xix, xxii, 41, 43, 46, 47, 52, 61–65, 70, 77, 78, 93, 96, 98–100, 102, 109, 115, 117, 118, 132, 133, 155, 156, 158–166, 194, 195, 197–199  
 Event Definition, 46, 64, 65  
 Event Flow, xix, 5, 9, 65, 156  
 Event Sub-Process, 160, 162, 165  
 Event-Based Gateway, 40, 52, 53, 110, 157  
 Evolving Algebras, 3, 9, 153  
 Exclusive Gateway, 11, 34–37, 39, 50, 76, 77, 160, 165, 166  
 Exclusive Instantiating Event-Based Gateway, 52, 53  
 Expression, 48, 64, 67, 81, 98, 108, 156, 158, 176, 195  
  
 Flow Element, 73  
 Flow Node, xix, 3, 5, 10–12, 34–36, 41, 44, 47, 50, 61, 64, 69–74, 76, 77, 79, 81, 83–89, 92–96, 98, 100, 102, 105, 108, 113, 115, 121, 122, 124, 127, 131–135, 153, 155, 160, 166, 175–177, 179–185, 187–190, 192, 194, 195, 197–199, 202, 205  
 Flow Object, 157  
 forall, 21  
 Function and rule body, 16, 20  
  
 Gateways, 3, 10–12, 34–39, 44–46, 48–52, 60, 62, 63, 69, 70, 75, 76, 86, 104–106, 110, 125, 126, 131, 157, 158, 161, 162, 165, 183, 201–203  
 Global Task, 155  
 Ground Model, xix, xxi, 1, 2, 4–6, 9, 15, 16, 18, 34, 41, 56, 76, 77, 79, 81, 93, 94, 96, 108, 110–114, 126, 127, 131, 132, 134, 135, 147, 153, 155, 160, 165, 190, 191, 213, 214, 219  
 Ground Model Method, 16  
 Guard, 20, 21  
  
 Inclusive Gateway, 35–37, 39, 63, 107, 121, 124, 125, 167, 179, 204  
 Inhibiting Path, 5, 121, 124, 167, 179  
 Instance, 5, 25, 26, 36, 47, 52, 65, 66, 70, 71, 73, 74, 76–78, 81, 83, 85, 86, 94–96, 98, 102, 103, 105, 106, 114, 118, 120, 132, 156, 159, 163, 165, 166, 175–177, 180, 182, 183, 191–195, 198, 200, 202, 205, 208, 210, 211  
 Intermediate Event, 40, 45, 47, 53, 62, 114, 154, 155  
 Item Definition, 40  
  
 Knowledge, 7  
 Knowledge Work, 7  
 Knowledge Worker, 7, 161  
  
 Link Trigger, 65, 158  
 Location, 16, 17, 19–21, 78, 94, 167, 191  
  
 Main Rule, 216, 217  
 Merging Gateway, 11, 51, 54  
 Message, xix, xxii, 5, 39, 40, 46, 47, 63, 116, 117, 133, 135, 163, 164, 205  
 Message Flow, 66, 112, 117, 133, 156  
 Message Notification, 67, 205  
 Message Trigger, 65, 67, 68, 109  
 Mixed Gateway, 52  
 Monitored (in) functions, 94, 95, 191, 210, 222  
  
 Notification, 47, 64–67, 114–118, 133–135, 158, 161, 163, 164, 166, 205, 206  
  
 par, 21  
 Parallel Gateway, 11, 12, 35, 37–39, 44, 49, 52, 53, 70  
 Parallel Instantiating Event-Based Gateway, 52  
 parblock, 21, 162

Pool, 46, 66  
 Process, xix, 7, 11, 25, 26, 39, 40, 46, 47, 50, 52, 53, 62, 64–66, 69, 71, 73, 77, 81, 86, 87, 105, 110, 111, 113, 114, 116, 118, 121, 125, 126, 133–135, 153, 155–167, 176, 182, 184, 202, 205, 209–211  
 Process Execution Conformance, 9, 10, 69, 117  
 Product Owner, 30  
 Propagation Resolution, 67, 207  
 Publication Resolution, 46, 66, 117, 207  
  
 Receive Task, 40, 61–63  
 Refinement Step, 29  
 Resource, 9, 93  
 Root Context, 114, 118, 156, 167  
 Rule, 16, 17, 19–22, 25–27, 29, 71, 73, 76, 81, 84–89, 92–94, 96, 100, 102, 103, 105, 106, 112, 116, 117, 120–122, 126, 154, 164, 176, 177, 181–185, 187–191, 193, 194, 197–204, 206–208, 210, 211, 213–217, 220, 221, 223–226, 229  
  
 Send Task, 40, 61–63  
 seq, 16, 21  
 seqblock, 16, 21, 162  
 Sequence Flow, 11, 25, 34, 35, 38, 39, 44, 45, 48, 49, 51, 52, 61, 64, 70–74, 76, 81, 83–87, 89, 91, 95, 104, 108, 110, 121, 124–126, 153, 156, 157, 160–162, 166, 175, 177, 179–184, 187, 188, 191, 201  
 Service Operation, 62, 116  
 Service Task, 40, 41, 62  
 Set, 70, 71, 73, 74, 78, 121  
 Signal, xix, xxii, 46, 47, 116, 117, 133, 135  
 Signal Notification, 67, 205  
 Signal Trigger, 67, 68, 109  
 Signature, 20, 21, 26, 164, 215  
 Sole Exclusive Gateway, 34, 70, 76, 131  
 Splitting Gateway, 11, 51, 122, 178  
 Start Event, 44, 52, 53, 62, 77, 78, 102, 166, 199, 205  
 State, 16, 19, 63, 154, 163  
 State Diagram, 8  
 Static Context, 113, 114, 118, 156, 208  
 Static functions, 48, 92, 98, 104, 108–110, 175, 194, 195, 201  
 Step, 16, 76  
 Stepwise Refinement Method, 4, 16, 33, 160, 163  
 Sub Context, 114  
 Sub-Process, 42, 44, 45, 73, 113, 153, 155, 158, 160–162, 166, 189  
 Sub-Process Start Event, 43, 44  
  
 Task, 39, 61–63, 114, 160, 163–165, 167  
 Technical level of ASM abstraction, xix, 15, 18, 19, 23, 26–30, 65, 78, 79, 114, 126, 132–135  
 Terminate Trigger, 133  
 Throw Event, 62, 63, 66, 100, 102, 109, 160, 198  
 Timer Trigger, 48, 67, 116, 158, 208  
 Token, 5, 25, 26, 35–38, 52, 63, 65, 70–76, 78, 81, 83, 85–89, 91, 92, 94–96, 100, 104, 106, 113, 114, 121, 122, 124, 125, 153, 156, 157, 160, 165–167, 177, 179, 180, 182–188, 191, 194, 196, 201, 202  
 Top-Level Process, 66, 95, 96, 113, 120, 156, 160, 166, 192, 194, 210, 211  
 Transaction Sub-Process, 66, 155  
 4-step Transition Approach, 24, 29, 79, 126, 127, 132, 135  
 Transition Rule, 20, 75, 132  
 Trigger, 46–48, 64, 65, 67, 99, 100, 102, 109, 114, 115, 120, 132, 158, 161, 164, 166, 195, 197, 198, 211  
  
 Uncontrolled Sequence Flow, 35, 37, 38  
 Universe, 18, 19, 28, 49, 56, 79, 127, 135, 167  
 Update, 16, 17, 19–21  
 Upstream Token, xix, xxi, xxii, 5, 87, 89, 114, 121, 122, 124, 125, 134,

179, 184, 185  
User Task, 39  
Vicious Circle, 125  
Workflows, 3, 4, 8–10, 15, 34, 113, 124,  
131, 153, 167



# Acronyms

**AD** Activity Diagram. 3, 8

**ASM** Abstract State Machine. xix, xxi, 1–6, 9, 10, 13, 15–19, 21, 23, 24, 26, 31, 69, 111–113, 132–135, 147, 149, 153–155, 157–159, 161–165, 167, 213–215, 218–220, 223–227

**BPEL** Business Process Execution Language. 8–10, 117, 147, 154, 155

**BPM** Business Process Modeling. 7, 9, 10

**BPMN** Business Process Model and Notation. xix, xxi, 1–6, 8–12, 15, 16, 24, 31, 33, 34, 36, 40–43, 46–48, 50, 52, 54–56, 61, 62, 64–66, 69–73, 75–77, 79, 83, 84, 88, 89, 93, 94, 96, 104, 108, 110–113, 116, 120, 127, 131, 133–135, 154–156, 159, 160, 163, 165, 166, 180, 181, 185, 188, 190, 191, 201

**BPMN<sub>e</sub>** Enhanced Business Process Model and Notation. 5, 6, 31, 35, 55, 61

**CM** Case Management. 8

**CMMN** Case Management Model and Notation. 8

**FSM** Finite State Machine. 16

**FWF** Austrian Science Fund. 4

**IEC** International Electrotechnical Commission. 3, 8, 10, 33

**ISO** International Standard Organization. 3, 8, 10, 33

**ITIL** Information Technology Infrastructure Library. 23

**JVM** Java Virtual Machine. 4, 6, 112

**OASIS** Organization for the Advancement of Structured Information Standards. 8

**OMG** Object Management Group. 8, 9

**PD** Process Diagram. 1, 10, 49, 64, 69, 120

**PN** Petri Nets. 3, 8–10

**RFC** Request for Change. 26, 30

**TA** Target Architecture. 2, 4, 6, 135

**UML** Unified Modeling Language. 8

**URI** Uniform Resource Identifier. 64

**WFE** Workflow Engine. xix, xxi, 3, 6, 9–11, 39, 65, 111–114, 116, 117, 120, 133–135, 157, 160, 163, 164, 167, 208

**WFI** Workflow Interpreter. xix, xxi, 1, 2, 5, 6, 77, 81, 110–114, 116, 117, 132, 133, 135, 156, 159, 176, 209–211

**WfM** Workflow Management. 7

**WM** Work Management. 1, 7

**WS** Web Service. 8, 164

**XML** Extensible Markup Language. 120

**YAWL** Yet Another Workflow Language. 9

# Glossary

## Symbols

**ASM ground model  $\text{\LaTeX}$  package** is a  $\text{\LaTeX}$  package allowing embedding of ASM code into  $\text{\LaTeX}$  documents. See Appendix C for detailed information. 56, 78, 133, 147, 213, 215, 218, 219, 223, 226, 229

**4-step Transition Approach** is an example approach developed to support ASM refinements and transitions between different abstraction levels in a system specification. 24, 29, 79, 126, 127, 132, 135, 149

## A

**Activity** is work that is performed within a process [1, sec. 10.2]. 3, 7, 8, 11, 12, 25, 26, 34–44, 46, 48, 57, 62, 65–67, 70, 76, 77, 79, 86, 93–96, 98, 102, 108, 113, 114, 118, 127, 128, 131–133, 147, 153–157, 159, 160, 162, 163, 165–167, 176, 183, 189, 191–194, 199, 209, *see* process & flow node

**Activity Diagram** is a set of actions, decisions, splits and joins with an initial state and an final state connected with arrows forming a workflow [69]. 3, 8, 151

**Actor** is a human or artificial task force, which performs a well structured work. Such work is usually highly repeatable and with high detail of specification.. 7, 39, 147, 167

**Ad-Hoc Sub-Process** is a specialized type of sub-process that is a group of activities that have no required sequence relationships. A set of activities can be defined for the process, but the sequence and number of performances for the activities is determined by the performers of the activities [1, sec. 10.2.5]. *see* sub-process & process

**Alternative Path** is a set of paths, build by distinct sequence flows, where only a subset of these paths may be choosen for tokens to traverse trough. 35, 37, 38, 76, 89, 121, 122, 147, 158–160, 178, 179, 188, *see* parallel path

**Anti-Inhibiting Path** is a path from a sequence flow containing a token to an non-empty incoming sequence flow (i.e. containing a tokens) of a flow node such that the path does not visit the flow node [60]. 5, 121, 122, 124, 125, 147, 167, 179, *see* inhibiting path & upstream token

**ASM Method** is a formal method, formerly called evolving algebras, described in [2], enabling high-level system design. xix, xxi, 16, 147, *see* evolving algebras

**Association** is used to associate information and artifacts with flow objects. Text and graphical non-flow objects can be associated with the flow objects

and flow. It is also used to show the activity used for compensation [1, sec. 8.3.1]. 147, 156, *see* connecting object, flow object & activity

## B

**Base Element** is the abstract super class for most BPMN elements. It provides the attributes id and documentation, which other elements will inherit [1, sec. 8.2.1]. 46–48, 64, 66, 67

**Basic functions** are functions, which are taken for granted (declared as “given”, typically those forming the basic signature) [2]. 16, 17, 19, 20, 24–29, 34, 78, 79, 81, 92, 94, 96, 98, 104, 108, 126–130, 133, 147, 161, 175, 176, 213–217, 219–221, 223–226, 229, *see* location, static function, dynamic function & derived function

**Dynamic functions** are functions that change as consequence of updates of the any machine or environment. Those functions represent the state [2]. *see* static function

**Controlled functions** are functions directly updateable only by the rules of the machine in context (i.e. in the context of the environment or another machine those functions must be monitored) [2]. 19, 81, 95, 98, 105, 147, 191, 192, 195, 202, 216, 217, *see* monitored function, out function & shared function

**Out functions** are functions updateable but not readable by the machine in context, and are readable but not updateable by the environment or another machine (i.e. in context of the environment or another machine those functions must be monitored) [2]. 19, 216, 217, *see* controlled function, monitored function & shared function

**Shared functions** are functions meant for interaction. Such functions can be updated by both: the machine in context, and the environment or another machine [2]. 19, 216, 217, *see* controlled function, monitored function & out function

**Static functions** are functions that never change during the run of the machine. Their values do not contribute to the state of the machine in context [2]. 19, 48, 81, 92, 98, 104, 108–110, 149, 175, 194, 195, 201, 216, 217, *see* dynamic function

**Basic Modeling Elements** are in [1, tab. 7.1] a set of simple modeling elements. 79, *see* activity, association, event, gateway, message, message flow, pool & sequence flow

**Function and rule body** is in ASM a body of a rule or derived function to be executed when such rule is fired or derived function called [2]. 16, 20, 26, 28, 148, *see* derived function & rule

**Boundary Event** represents exception or compensation handling by placing the intermediate event on the boundary of an activity [1, sec. 10.4.4]. 42, 43, 47, 48, 65, 77, 98, 104, 132, 147, 156, 163, 165, 194, 201, *see* activity & intermediate event

**BPEL Process Execution Conformance** is one of the four conformances defining applicable matching compliance points necessary for a software to claim

BPMN 2.0 BPEL Process Execution Conformance. The applicable compliance points may be found in [1, sec. 2.3]. 9, 10, 117, 147

**BPMN<sup>+</sup>**. 6, 147

**Business level of ASM abstraction** of an ground model is targeting *technically aware business people* and is the starting abstraction level during the desing process. Usually one of the version on this level of abstraction may serve as a succinct process-oriented model of the to-be-implemented piece of *real world*, transparent for both the customer and the software designer so that it can serve as the basis for the software contract [15]. xix, 15, 18, 19, 23, 24, 26–30, 55, 65, 78–80, 113, 114, 126, 132, 133, 135, 147, 169, 215, *see* technical level

**Business Process** 33, 69, *see* process

**Business Rule Task** provides a mechanism for the process to provide input to a *Business Rules Engine* and to get the output of calculations that the *Business Rules Engine* might provide [1, sec. 10.2.3.1]. 40, *see* task

## C

**Call Activity** identifies a point in the process where a global process or a global task is used. IT acts as a “wrapper” for the invocation of a global process or global task within the execution [1, sec. 10.2.6]. 113, 147, 160, *see* activity

**Cancel Triggers** are only used in the context of modeling transaction sub-processes. There are two variations: a catch intermediate event and an end event [1, sec. 10.4.5]. 46, 48, 65–67, 132, 147, *see* trigger

**Cancellation Resolution** triggers a compensation handler of all completed activities inside the canceled activity and terminates all running activities. If the canceled activity is a transaction sub-process sub-process it will be rolled back [1]. 66, *see* cancel trigger & transaction sub-process

**Case** is a top-level concept that combines all elements that constitute a case model [71, sec. 5.2]. 8, 147, 155

**Catch Event** is an event, which can be activated with a occurence of defined triggers inside the event flow node [1]. 40–43, 46, 47, 59, 62, 63, 65–67, 77, 98, 100, 109, 114, 116, 117, 132, 147, 155, 158, 161, 163, 164, 194, 197, 205, *see* event

**choose** is an ASM construct, which executes with an arbitrary element chosen among the provided set satisfying a selection property [2]. 21, 147

**Choreography** is a type of a process, but differs in purpose and behavior from a standard BPMN process. It formalizes the way business participants coordinate their interactions. The focus is not on orchestrations of the work performed within these participants, but rather on the exchange of information (messages) between these participants [1, ch. 11]. 9, 147

**Collaboration** package contains classes that are used for modeling collaborations, which is a collection of participants shown as pools, their interactions as shown by message flows, and may include processes [1, ch. 9]. 9, 117, 134, 147, 162

**Collapsed Sub-Process** is a sub-process that hides its details [1, sec. 10.2.5]. *see* sub-process

**Compensation Activity** is connected to the boundary event through an association defining a compensation handler for the activity the boundary event is connected to [1, sec. 10.6.1]. *see* activity & boundary event

**Compensation Resolution** triggers a compensation handler of the activity in the instance in which it was fired [1]. 65, *see* compensation trigger

**Compensation Triggers** are used in the context of triggering or handling compensation [1, sec. 10.4.5]. 43, 46, 48, 65–67, 147, *see* trigger

**Complex Gateway** can be used to model complex synchronization behavior [1, sec. 10.5.5]. 5, 36, 48, 50–52, 63, 73, 75, 87, 91, 92, 105–108, 121, 122, 125, 134, 147, 157, 177, 179, 185, 188, 202, 204, *see* gateway

**Conditional Sequence Flow** is a sequence flow, which defines a condition expression indicating that a token will be passed down the sequence flow only if the expression evaluates to `true` [1, sec. 8.3.13]. 35, 36, 38, 39, 49, 62–64, 70–72, 83, 108, 110, 147, 180, *see* sequence flow

**Conditional Trigger** 46, 48, 59, 65, 67, 98, 114, 116, 147, 158, 195, 208, *see* trigger

**Connecting Object** are connecting flow nodes to each other or to other objects [1, sec. 7.2]. 156, *see* event flow, message flow & sequence flow

**Event Flow** represents in BPMN 2.0 an abstract flow for events and is modeled only using event flow nodes and triggers without any connecting objects [1]. xix, 5, 9, 65, 148, 156, *see* connecting object

**Constraint** is a condition that the solution must satisfy or a mechanism, which frequently is used also to impose desired properties on functions appearing in the signature of a machine [2]. 7, 15, 62, 76, 78, 89, 93, 134, 147, 187, *see* guard

**Context** 5, 114–116, 118, 127, 130, 133–135, 147, 156, 205, 208, 209, *see* context tree

**Root Context** is a context created for each new processes instance. The environment can communicate with such instance by sending notifications to it. 113, 114, 118, 149, 156, 167, *see* context tree

**Static Context** exists only once in a instance of a WFI and is created as soon as the WFI is started. All existing processes expose here their top-level start events, which are those with no defined trigger or with “Message”, “Timer”, “Conditional” or “Signal” trigger [1], waiting for a corresponding *notification* to fire them. 113, 114, 118, 149, 156, 208, *see* context tree

**Sub Context** is a context created for every new activity instance inside a running process instance. This context is not visible to the environment but may populate some uncached events to the parent context. Also notifications sent by the environment to the root context may be populated down to sub contexts. 113, 114, 149, 156, *see* context tree

**Context Tree** is a tree structure build by contexts for a WFI instance. The root of this tree is a static context with its immediate children being root contexts for each top-level process instance running in the WFI. The rest of the tree is build by sub contexts. 113, 147, 158, 161, 164, *see* context, static context, root context & sub context

**Control Flow** is the primary flow in BPMN next to others, e.g., message flow or event flow. Sequence flows are used to model a control flows in a process [1]. xix, 5, 9, 34, 37, 40, 44, 49, 51–53, 69, 70, 77, 89, 111, 112, 131, 132, 147, 156, 165,

188, *see* sequence flow

**Controlled Sequence Flow** is a flow that proceeds from one flow object to another, via a sequence flow link, but is subject to either conditions or dependencies from other flow as defined by a gateway. Typically, this is seen as a sequence flow between two activities, with a conditional indicator (mini-diamond) or a sequence flow connected to a gateway [1]. 36, 37, 71, 147, *see* sequence flow

**CoreASM Engine** is a project focusin on the design of a lean executable ASM language, in combination with a supporting tool environment for high-level design, experimental validation and formal verification of abstract system models. This project can be accessed at <http://www.coreasm.org>. The CoreASM project is an Open Source project licensed under the Academic Free License version 3.0. 4, 16, 147, *see* ASM method

## D

**Data Association** are used to move data between data objects, properties, and inputs and outputs of activities, processes, and global tasks. tokens do not flow along it, and as a result they have no direct effect on the flow of the process [1, sec. 10.3.1]. *see* association & connecting object

**Data Input** Activities and processes often need data in order to execute. In addition they can produce data during or as a result of execution. Data requirements are captured as data inputs and input sets. Data that is produced is captured using data outputs and output sets [1, p. 211]. 40, 147, 157, 160, 162, *see* data output, input set & output set

**Data Output** Activities and processes often need data in order to execute. In addition they can produce data during or as a result of execution. Data requirements are captured as data inputs and input sets. Data that is produced is captured using data outputs and output sets [1, p. 211]. 157, 160, 162, *see* data input, input set & output set

**Data-Based Gateway** is a non-event-based gateway [1, sec. 10.5]. 50–52, 62, 63, 75, 76, 147, *see* gateway & event-based gateway

**Default Expression** is used as a constant expression, which evaluates to true if no other expression in the same provided set holds. 64, 147, *see* expression, default flow, exclusive gateway, inclusive gateway & complex gateway

**Default Sequence Flow** is a sequence flow that has an exclusive, inclusive, or complex gateway or an activity as its source and is defined with it as default. Such sequence flow is taken (a token is passed) only if all the other outgoing sequence flows from the activity or gateway are not valid, i.e., their condition expressions evaluate to `false`) [1, sec. 8.3.13]. 36, 64, 108, 147, *see* sequence flow

**Deployment Manager** is a part of the WFE responsible for deploying new processes so they can be executed. 120, 147

**Derived functions** are functions, which are not updateable either by the machine in context nor by the environment nor by any other machine, but may be read by any of those. They yield values, which are defined by a fixed

scheme terms of other functions [2]. 16, 17, 19–21, 24–26, 28, 34, 79, 81, 83, 92–94, 98, 99, 105, 113, 127, 147, 154, 165, 175–177, 180, 189, 192, 195, 202, 215–217, 221, *see* basic function

**Direct Resolution** is a event resolution technique used for all implicit triggers (conditional trigger and timer trigger) and for link trigger. Such triggers do not use notification concept and their resolution is dependent only on static or dynamic data evaluation, e.g. expression or linked event nodes [1]. 65, *see* timer trigger & conditional trigger

## E

**End Event** indicates where a process will end. In terms of sequence flows, it ends the flow of the process, and thus, will not have any outgoing sequence flows [1, sec. 10.4.3]. 41–47, 52, 77, 78, 102, 132, 147, 155, 199, *see* event

**Environment** in ASMs is another ASM machine representing whether a context where the designed system is running or another agent in case of multi-agent systems [2]. 19, 147, 154, 157

**Error** represents the content of an error event or the fault of a failed operation [1, sec. 8.3.3]. 46–48, 62, 65, 118, 133, 147, *see* error trigger & error notification

**Error Notification** is a notification traversing the context tree and searching for an event with appropriate trigger. 66, 67, 118, 147, 204, 205, *see* context tree, error trigger & notification

**Error Trigger** defines the details of the error which happened in case of throw events or which should be handled in case of catch events [1]. 46–48, 65, 67, 68, 109, 118, 148, 158, 198, *see* trigger & error

**Escalation** identifies a business situation that a process might need to react to [1, sec. 8.3.4]. 46–48, 62, 65, 118, 133, 148, *see* escalation trigger & escalation notification

**Escalation Notification** is a notification traversing the context tree and searching for an event with appropriate trigger. 66, 67, 148, 204, 205, *see* notification & escalation trigger

**Escalation Trigger** defines the details of the escalation which happened in case of throw events or which should be handled in case of catch events [1]. 46, 47, 67, 68, 109, 118, 148, 198, *see* trigger & escalation

**Event** is something that “happens” during the course of a process. These events affect the flow of the process and usually have a cause or an impact and in general require or allow for a reaction [1, sec. 10.4]. xix, xxii, 7, 34, 36, 41–44, 46–49, 52, 53, 57, 61–65, 67, 70, 76–79, 92, 93, 96, 98–100, 102, 109, 113–115, 117, 118, 127, 129, 131–133, 148, 155, 156, 158–166, 189, 194–199, *see* flow node

**Event Definition** 46, 64, 65, 148, *see* trigger

**Event Sub-Process** is a specialized sub-process that is used within a process (or sub-process) and triggered by an event [1, sec. 10.2.5]. 148, 160, 162, 165, *see* sub-process, process & event

**Event-Based Gateway** represents a branching point in the process, where the alternative paths that follow the gateway are based on events that occur, rather



than the evaluation of *Expressions* using process data [1, sec. 10.5.6]. 5, 40, 41, 48, 49, 51–53, 62, 63, 91, 110, 135, 148, 157, 188, *see* gateway & event

**Evolving Algebras** is a formalization method [101]. 3, 9, 148, 153, *see* ASM method

**Exclusive Event-Based Gateway** configuration is a race condition, where the first event that is triggered wins [1, sec. 10.5.6]. 52, *see* , event-based gateway & exclusive instantiating gateway

**Exclusive Gateway** is used to create alternative paths within a process flow. For a given instance of the process, only one of the paths can be taken [1, sec. 10.5.2]. 11, 34–39, 48, 50–52, 63, 72–77, 83, 86, 91, 105, 106, 108, 131, 148, 157, 160, 165, 166, 180, 183, 188, 202, 203, *see* gateway & alternative path

**Exclusive Instantiating Event-Based Gateway** is meant for situations, where the modeler wants the process to be instantiated by one of a set of events while still requiring all of the events for the working of the same process instance [1, sec. 10.5.6]. 52, 53, 148, *see* gateway, event, event-based gateway & exclusive event-based gateway

**Expression** is used to specify an natural-language text expression, which is not executable [1, sec. 8.3.6]. 46, 48, 64, 67, 81, 98, 108, 148, 156–158, 160, 176, 195, *see* complex gateway, conditional flow, conditional trigger, default expression, exclusive gateway, formal expression & inclusive gateway

**Extended Modeling Elements** is in [1, tab. 7.1] a set of complex and compound modeling elements. 11, 62

## F

**Flow Element** is the abstract super class for all elements that can appear in a process flow, which are flow nodes and sequence flows [1, sec. 8.3.7]. 11, 34, 52, 54, 62, 69, 70, 72, 73, 134, 148, 166

**Flow Node** is in BPMN an abstraction of activities, events and gateways [1]. xix, 3, 5, 10–12, 34–36, 41, 44, 47, 48, 50, 52, 61, 62, 64, 69–74, 76, 77, 79, 81, 83–89, 92–96, 98, 100, 102, 105, 108, 113, 115, 121, 122, 124, 127, 128, 131–135, 148, 153, 155, 156, 159, 160, 164, 166, 175–177, 179–185, 187–190, 192–199, 202, 205, *see* flow element, activity, event & gateway

**Flow Object** 53, 148, 153, 157, *see* flow element

**forall** is an ASM construct, which executes with all elements in the provided set satisfying a selection property [2]. 21, 148

**Formal Expression** is used to specify a formal expression, which is executable in a WFI if the language of the expression is known to the WFI. 64, *see* expression

## G

**Gateways** are used to control how sequence flows interact as they converge and diverge within a process [1, sec. 10.5]. xix, xxii, 3, 7, 8, 10–12, 34–39, 41, 44–46, 48–53, 60, 62–64, 69–72, 75, 76, 79, 84, 86, 93, 104–106, 110, 113, 125–127, 130, 131, 148, 157–159, 161, 162, 165, 181, 183, 189, 201–203, *see* flow node

**Global Task** is a reusable, atomic task definition that can be called from within any process by a call activity [1, sec. 10.2.7]. 148, 155, 157, *see* call activity & task

**Ground Model** is formal model (or blueprint) of a system [2]. xix, xxi, 1, 2, 4–6, 9, 15, 16, 18, 34, 41, 56, 76, 77, 79, 81, 93, 94, 96, 108, 110–114, 126, 127, 131, 132, 134, 135, 147, 148, 153, 155, 160, 165, 190, 191, 213, 214, 219

**Ground Model Method** a method encompassing requirement capture and stepwise refinement method to build a ground model [2]. 16, 148, *see* ground model

**Guard** is a `if guard then updates` statement used for guarded updates in transition rules [102]. 20, 21, 148, 163, *see* constraint, rule & update

## I

**Implicit Throw Event** is a non-graphical event, sub-type of throw event [1, sec. 10.4.1]. 42, 43, 77, 116, *see* throw event

**Inclusive Gateway** can be used to create alternative but also parallel paths within a process flow. Unlike the exclusive gateway, all condition expressions are evaluated [1, sec. 10.5.3]. xix, 5, 35–39, 48, 50–52, 63, 72–75, 83, 87, 91, 92, 105–108, 121, 122, 124–126, 134, 148, 157, 167, 177, 179, 180, 184, 188, 202–204, *see* alternative path, exclusive gateway & parallel path

**Inhibiting Path** a path from a sequence flow containing a token to an empty incoming sequence flow (i.e. not containing any tokens) of a flow node such that the path does not visit the flow node [60]. 5, 121, 122, 124, 125, 148, 167, 179, *see* anti-inhibiting path & upstream token

**Input Set** Activities and processes often need data in order to execute. In addition they can produce data during or as a result of execution. Data requirements are captured as data inputs and input sets. Data that is produced is captured using data outputs and output sets [1, p. 211]. 157, 162, *see* data input, data output & output set

**Instance** 5, 25, 26, 36, 47, 52, 65, 66, 70, 71, 73, 74, 76–78, 81, 83–86, 88, 89, 94–96, 98, 100, 102, 103, 105, 106, 113, 114, 118, 120, 132, 134, 135, 148, 156, 159, 160, 163, 165, 166, 175–177, 180–183, 185, 186, 190–195, 197, 198, 200, 202, 205, 208–211, *see* process, activity & token

**Instance Manager** is a part of the WFE responsible for managing top-level process instances. 114

**Intermediate Event** indicates where something happens somewhere between the start and end of a process. It will affect the flow of the process, but will not start or (directly) terminate the process [1, sec. 10.4.4]. 40–45, 47, 53, 62, 77, 114, 132, 148, 154, 155, *see* event

**Interrupting Event** denotes that the sub-process encompassing the event sub-process should be cancelled [1, tab. 10.87] in case such event is triggered. 41–43, 47, 132, *see* catch event, sub-process start event & boundary event

**Item Definition** is in BPMN an element that can specify an import reference where the proper definition of the structure is defined [1, sec. 8.3.10]. 40, 64, 148

## K

**Knowledge** 7, 148

**Knowledge Work** work using knowledge as a core to achieve the goal. Knowledge is seen here as a hardly describable or structurable skill of a knowledge worker such as creativity. 7, 148, 161

**Knowledge Worker** as a opposite to a manual worker [103]. Knowledge worker performs his/her work using knowledge [61]. 7, 148, 161, *see* knowledge work

## L

**Link Intermediate Event** is a mechanism for connecting two sections of a process. Link events can be used to create looping situations or to avoid long sequence flow lines. Link event uses are limited to a single process level (i.e., they cannot link a parent process with a sub-process). Paired Intermediate Events can also be used as “Off-Page Connectors” for printing a process across multiple pages. They can also be used as generic “Go To” objects within the process level. There can be multiple source link events, but there can only be one target link event [1, tab. 10.89]. 135, *see* intermediate event

**Link Trigger** 46, 48, 65, 67, 133, 148, 158, *see* link event

**Location** is in ASMs is particular combination of arguments used with a basic function [2]. 16–21, 34, 78, 94, 148, 167, 191, *see* basic function

## M

**Merging Gateway** is a gateway, which have multiple incoming, but only one outgoing sequence flows [1, sec. 10.5.1]. 11, 37, 49–52, 54, 60, 70, 72, 73, 75, 76, 83, 86, 127, 131, 148, 182, *see* gateway

**Message** represents the content of a communication between two participants [1, sec. 8.3.11]. xix, xxii, 5, 39, 40, 46, 47, 63, 65, 113, 116, 117, 133, 135, 148, 155, 161, 163, 164, 205, 206, *see* message trigger & message notification

**Message Flow** is used to show the flow of messages between two participants that are prepared to send and receive them [1, sec. 9.3]. 66, 112, 117, 133, 148, 155, 156, *see* connecting object

**Message Notification** is a notification traversing the context tree and searching for an event with appropriate trigger. 66, 67, 148, 205, *see* notification & message trigger

**Message Trigger** defines the details of the message event, which is sent in case of throw events or which waited for in case of catch events [1]. 40, 41, 46, 47, 59, 61–63, 65–68, 109, 114, 148, 163, *see* trigger & message

**Mixed Gateway** is a gateway, which have both multiple incoming and outgoing sequence flows [1, sec. 10.5.1]. 49, 51, 52, 148, *see* gateway

**Multiple Event** *see* multiple trigger

**Multiple Trigger** enables multiple ways of ways of the process or multiple consequences of ending the process [1, sec. 10.4.5]. 46, *see* trigger

## N

**Non-Interrupting Event** denotes that the sub-process encompassing the event sub-process should not be cancelled [1, tab. 10.87] in case such event is triggered. *see* catch event, sub-process start event & boundary event

**None Trigger** 46, *see* trigger

**Notification** is a materialization of “something happened”, represented by a notification object defining what happened and additional information, e.g., timestamp or payload. Such an object has to reach an event node so it can fire in a similar way as tokens do in sequence flows.. xix, xxii, 5, 47, 64–67, 109, 113–118, 127, 130, 133–135, 148, 156, 158, 161, 163, 164, 166, 204–206, 208

## O

**Output Set** Activities and processes often need data in order to execute. In addition they can produce data during or as a result of execution. Data requirements are captured as data inputs and input sets. Data that is produced is captured using data outputs and output sets [1, p. 211]. 157, 160, *see* data input, data output & input set

## P

**par** construct is in ASMs an explicit definition that two statements will be executed in parallel [2]. 21, 148, *see* parblock, seq & seqblock

**Parallel Gateway** is used to synchronize (combine) parallel flows and to create parallel flows [1, sec. 10.5.4]. 11, 12, 35, 37–39, 44, 48–53, 63, 70, 72–74, 83, 86, 105, 106, 148, 180, 183, 202, 203

**Parallel Instantiating Event-Based Gateway** represents a case, even when the first event is triggered and the process is instantiated, the other events of the gateway configuration are not disabled. The other events are still waiting and are expected to be triggered before the process can (normally) complete [1, sec. 10.5.6]. 52, 53, 148, *see* gateway, event, event-based gateway & exclusive instantiating gateway

**Parallel Multiple Trigger** defines the case where multiple triggers required. All of them are necessary [1, sec. 10.4.5]. 46, *see* trigger

**Parallel Path** is a set of paths, build by distinct sequence flows, where all of these paths have to be choosen for tokens to traverse trough. 11, 35, 37, 38, 44, 53, 77, 89, 102, 160, 188, 199, *see* alternative path

**parblock** construct is in ASMs an explicit definition that all statements inside the parblock will be executed in parallel. If no seqblock or parblock is defined statements will be executed in parallle as this is the default behavior of ASMs [2]. 21, 148, 162, *see* par, seq & seqblock

**Passing Gateway** is a gateway, which have only one incoming and one outgoing sequence flow. *see* gateway

**Pool** is the graphical representation of a participant in a collaboration. 46, 65, 66, 149, 155, 163, *see* collaboration

**Process** describes a sequence or flow of activities in an organization with the objective of carrying out work [1, ch. 10]. xix, 7, 8, 10, 11, 25, 26, 34, 39, 40, 46,

47, 50, 52, 53, 61, 62, 64–66, 69, 71, 73, 77, 81, 86, 87, 105, 110, 111, 113, 114, 116–118, 120, 121, 125, 126, 131, 133–135, 149, 153, 155–167, 176, 182, 184, 202, 205, 209–211, *see* activity

**Process Execution Conformance** is one of the four types of conformance defining applicable matching compliance points necessary for a software to claim BPMN 2.0 Process Execution Conformance. The applicable compliance points may be found in [1, sec. 2.2]. 9, 10, 69, 117, 149

**Product Owner** The Product Owner represents the stakeholders and is the voice of the customer. He or she is accountable for ensuring that the team delivers value to the business. 30, 149

**Propagation Resolution** is a event resolution technique used for forwarding notifications up the process instance hierarchy tree. The corresponding notifications are forwarded from the location they were triggered to the innermost enclosing scope instance up the process instance hierarchy tree. Such enclosing scope is generally represented by an activity with an attached boundary event able to catch the notification. If no such catch event is found and the notification reaches the root of the process instance tree, such notification will remain unresolved. It is then up to the WFE how such situation will be handled. 65, 67, 113, 118, 149, 207, *see* escalation trigger, error trigger, escalation, error, publication & boundary event

**Publication Resolution** is a event resolution technique used for forwarding notifications down the process instance hierarchy tree. It allows the notification to be caught by any catch event in any scope. This is used for message and signal triggers used to communicate between pools and processes. 46, 65, 66, 113, 117, 149, 207, *see* message trigger, signal trigger, message, signal & propagation

## R

**Receive Task** is a simple task that is designed to wait for a message to arrive. Once the message has been received, the task will transit to the “Completed” life-cycle state [1, sec. 10.2.3.1]. 39–41, 61–63, 149, *see* task & life-cycle state

**Refinement Step** is one step in the stepwise refinement method [2]. 16, 24–26, 29, 132, 133, 149, *see* stepwise refinement method

**Resource** The *Resource* class in the BPMN meta-model is used to specify resources that can be referenced by activities. These resources can be human resources as well as any other resources assigned to activities during process execution [1]. 7, 9, 93, 94, 149, 190, 191

**Root Element** is the abstract super class for all BPMN elements that are contained within definitions [1, sec. 8.2.5]. 46, 47, 66, 67

**Rule** is an ASM construct is a transition rule [83] in the ASM method yielding possible empty set of guarded updates resulting in a new state. 16, 17, 19–22, 24–29, 41, 71, 73, 76–79, 81, 84–89, 92–94, 96, 98, 100, 102–106, 108, 112, 113, 116, 117, 120–122, 126–130, 133, 149, 154, 160, 164, 165, 175–177, 181–185, 187–191, 193, 194, 197–204, 206–208, 210, 211, 213–217, 219–221, 223–226, 229, *see* guard, state & update

**Main Rule** is an ASM rule used as a signature of a whole ASM machine or module. This rule is fired if the machine or module is called [2]. 148, 216, 217, *see* signature

**Sub Rule** 85, 181

## S

**Script Task** is executed by a WFE where the defined a script in the defined language that the WFE can interpret will be executed. When the task is in the “Ready” life-cycle state, the WFE will execute the script and when the script is completed, the task will transit to the “Completed” life-cycle state [1, sec. 10.2.3.1]. 39, 40, *see* task & life-cycle state

**Send Task** is a simple task that is designed to send a message. Once the message has been sent, the task will transit to the “Completed” life-cycle state [1, sec. 10.2.3.1]. 39–41, 61–63, 149, *see* task & life-cycle state

**seq construct** is in ASMs an explicit definition that two statements will be executed sequentially in order [2]. 16, 21, 149, *see* seqblock, par & parblock

**seqblock construct** is in ASMs an explicit definition that all statements inside the seqblock will be executed sequentially in order they are defined [2]. 16, 21, 149, 162, *see* seq, par & parblock

**Sequence Flow** is used to show the order of flow nodes in a process [1, sec. 8.3.13]. 3, 10–12, 25, 34–39, 41, 44, 45, 47–52, 61–65, 69–76, 78, 81, 83–89, 91–93, 95, 100, 102, 104–106, 108, 110, 114, 121, 124–127, 131, 133, 134, 149, 153, 156–162, 165, 166, 175, 177, 179–185, 187–191, 196, 199, 201–203, *see* connecting object

**Service Operation** defines messages that are consumed and, optionally, produced when the operation is called. It can also define zero or more errors that are returned when operation fails [1, sec. 8.4.3]. 62, 116, 149

**Service Task** is a task that uses some sort of service, which could be a WS or an automated application [1, sec. 10.2.3.1]. 39–41, 62, 149, *see* task

**Set** is a mathematical structure based on the set theory used on higher levels of abstraction as an alternative to a type system [80, 82]. 70, 71, 73, 74, 78, 121, 149, *see* universe

**Signal** represents the type of signalization between two participants. xix, xxii, 46, 47, 65, 113, 116, 117, 133, 135, 149, 205, 206, *see* signal trigger & signal notification

**Signal Notification** is a notification traversing the context tree and searching for an event with appropriate trigger. 66, 67, 149, 205, *see* notification & signal trigger

**Signal Trigger** defines the details of the signal event, which is sent in case of throw events or which waited for in case of catch events [1]. 46, 47, 59, 65–68, 109, 114, 149, 163, *see* trigger & signal

**Signature** of a rule or function comes with the name of the rule or function, indication of arity, possibly with the types/universes of the arguments [83]. 19–21, 26, 29, 149, 164, 213, 215, *see* basic function, derived function & rule

**Declaration Signature** used for declaring rules or functions and their parameter universes and the return value universe for functions. 25, 26, 79, 126, 147, 215, 217–220, 224, *see* basic function, derived function & rule

**Definition Signature** used for defining, calling or referring to a rule or function. rules and derived functions may also define their bodies after this type of signature. 26, 147, 215, 217, 218, 221, *see* basic function, derived function & rule

**Sole Exclusive Gateway** is an extension of exclusive gateway in BPMN, which does not allow multiple tokens from the same instance on the incoming sequence flows. The original BPMN exclusive gateway allows such situation and produced for each such token from the same instance a distinct token on the outgoing side of the gateway. 34, 70, 76, 131, 149, *see* exclusive gateway

**Splitting Gateway** is a gateway, which have only one incoming, but multiple outgoing sequence flows [1, sec. 10.5.1]. 11, 49–52, 54, 60, 70, 74, 75, 83, 89, 122, 127, 131, 149, 178, 188, *see* gateway

**Start Event** indicates where a particular process will start. In terms of sequence flows, the Start Event starts the flow of the process, and thus, will not have any incoming sequence flows [1, sec. 10.4.2]. 41–44, 52, 53, 62, 77, 78, 102, 122, 132, 149, 166, 178, 199, 205, *see* event, top-level start event & sub-process start event

**State** 16, 19, 63, 149, 154, 163

**State Diagram** is a type of diagram used to describe the behavior of systems [69]. 8, 149

**Step** 16, 76, 149

**Stepwise Refinement Method** is a technical support for the crossing of abstraction levels in an ASM ground model [2]. 2, 4, 16, 24, 29, 33, 132, 133, 149, 160, 163, *see* ground model

**Sub-Process** is an activity whose internal details have been modeled using activities, gateways, events, and sequence flows [1, sec. 10.2.5]. 42, 44, 45, 73, 86, 92, 94, 95, 113, 149, 153, 155, 158, 160–162, 166, 182, 189, 190, 192, *see* activity & process

**Sub-Process Start Event** starts (instantiates) an inline event sub-process (see page [1, p. 176]). In that case, the same event types as for boundary events are allowed (see [1, tab. 10.86]). 42–44, 47, 132, 149

## T

**Task** is an atomic activity within a process flow. A task is used when the work in the process cannot be broken down to a finer level of detail [1, sec. 10.2.3]. 5, 7, 39, 40, 61–63, 94, 114, 135, 149, 160, 163–165, 167, 190, 191, *see* activity

**Technical level of ASM abstraction** is targeting more *technical people* and *implementers*. The emphasis passes from the natural text document to rather pseudo-code constructs. The complexity of the pseudo-code constructs increases and the amount of abstract functions and rules decreases. Types,

properties and relations in this abstraction level may be mostly defined. xix, 15, 18, 19, 23, 24, 26–30, 65, 78, 79, 114, 126–130, 132–135, 149, *see*

**Terminate Trigger** is a type of end, which indicates that all activities in the process should be immediately ended [1, sec. 10.4.3]. 46, 48, 65–67, 133, 149, 166, *see* trigger

**Termination Resolution** is bounded with a top-level process instance, which should be terminated, meaning that all running activities inside such process and instance should be ended without any compensation or any other handling. Terminate triggers do not have a corresponding notification object. 66, *see* terminate trigger & top-level process

**Throw Event** is an event, which throws a trigger defined inside the event flow node [1]. 40–43, 46, 47, 59, 62, 63, 65–67, 77, 100, 102, 109, 132, 149, 158, 160, 161, 164, 198, *see* event

**Time Expression** is used to specify a time expression replacing the `timeDate`, `timeCycle` and `timeDuration` from [1, tab. 10.101]. 46, 48, 59, 64, 67, *see* expression & timer trigger

**Timer Trigger** 46, 48, 59, 65, 67, 98, 114, 116, 149, 158, 195, 208, *see* conditional trigger & time expression

**Token** 3, 5, 25, 26, 35–38, 52, 63, 65, 70–76, 78, 81, 83–89, 91–96, 100, 104–106, 108, 113, 114, 121, 122, 124, 125, 149, 153, 156, 157, 160, 162, 165–167, 176, 177, 179–188, 190, 191, 194, 196, 201, 202

**Enabling Token** is a token, which is residing on a directly incoming sequence flow of a flow node. To such flow node the token is enabling [1]. 70, 81, 96, 147, 193

**Top-Level Process** is in BPMN a set of flow elements [1, ch. 10]. 66, 95, 96, 113, 120, 149, 156, 160, 166, 192, 194, 210, 211, *see* process

**Top-Level Start Event** starts (instantiates) a top-level process. The trigger for a start event is designed to show the general mechanisms that will instantiate that particular process [1, p. 240]. 42–44, 114, 156, *see* event

**Transaction Sub-Process** is a specialized type of sub-process that will have a special behavior that is controlled through a transaction protocol (such as WS-Transaction) [1, sec. 10.2.5]. 66, 149, 155, *see* sub-process & activity

**Transition Rule** 17, 20, 28–30, 70, 75, 131–133, 149, *see* rule

**Trigger** represents a definition of an event [1]. xix, xxii, 36, 46–48, 59, 64, 65, 67, 98–100, 102, 109, 113–115, 120, 127, 129, 132, 149, 155, 156, 158, 161, 162, 164, 166, 194, 195, 197, 198, 211, *see* event definition

## U

**Uncontrolled Sequence Flow** means that, for each token arriving on any incoming sequence flows into the flow node, the flow node will be enabled independently of the arrival of tokens on other incoming sequence flows. The presence of multiple incoming sequence flows behaves as an exclusive gateway [1, sec. 13.2.1]. 35, 37, 38, 149, *see* sequence flow & exclusive gateway



**Universe** 18–21, 25–29, 49, 55–60, 79, 80, 127, 135, 149, 167, 215, 216, 224, 227, *see* set

**Super universe** is an equivalent to a superset of a set for universe [2]. 56

**Update** in the ASM method updates a concrete location with a new value [2]. 16, 17, 19–21, 149, 154, 160, 163, *see* location & rule

**Upstream Token** is a token which has an inhibiting path but no anti-inhibiting path to an inclusive gateway [60]. xix, xxi, xxii, 5, 74, 87, 89, 113, 114, 121, 122, 124, 125, 134, 149, 179, 184, 185, *see* token

**User Task** is a task that is not managed by any WFE. It can be considered as an unmanaged task, unmanaged in the sense of that the WFE doesn't track the start and completion of such a task [1, sec. 10.2.4.1]. 39, 40, *see* task & life-cycle state

**User Task** is a typical workflow task, where a human actor performs the task with the assistance of a software application. The life-cycle of the task is managed by the WFE and is executed in the root context of a process [1, sec. 10.2.4.1]. 39, 40, 150, *see* task & life-cycle state

## V

**Vicious Circle** a situation in which an attempt to resolve one problem creates new problems that lead back to the original situation.

In *Logic*: A fallacy in reasoning in which the premise is used to prove the conclusion, and the conclusion used to prove the premise.

In (*Philosophy* / *Logic*) *Logic*:

- a.** a form of reasoning in which a conclusion is inferred from premises the truth of which cannot be established independently of that conclusion
- b.** an explanation given in terms that cannot be understood independently of that which was to be explained
- c.** a situation in which some statement is shown to entail its negation and vice versa, as *this statement is false* is true only if false and false only if true. 125, 150

## W

**Workflows** are well specified and highly repeatable patterns of business activity. It can be depicted as a sequence of activities, declared as work of an actor. 3, 4, 8–10, 15, 34, 113, 124, 126, 131, 150, 153, 167



## Appendix A

# Transition from business level

Here the transition from business level is shown.

$$\begin{aligned} & \text{defaultSequenceFlowOfActivity} \Rightarrow & (A.1) \\ \Rightarrow & \text{defaultSequenceFlow} : \text{ACTIVITIES} \end{aligned}$$

$$\begin{aligned} & \text{lifeCycleStateOfActivityInInstance} \Rightarrow & (A.2) \\ \Rightarrow & \text{lifeCycleState} : \text{ACTIVITIES} \times \text{INSTANCES} \end{aligned}$$

$$\begin{aligned} & \text{ActivationBehaviorOfFlowNode} \Rightarrow & (A.3) \\ \Rightarrow & \text{ActivationBehavior} : \text{FLOW\_NODES} \end{aligned}$$

$$\begin{aligned} & \text{ComplexMergeBehaviorOfFlowNode} \Rightarrow & (A.4) \\ \Rightarrow & \text{ComplexMergeBehavior} : \text{FLOW\_NODES} \end{aligned}$$

(A.5)

$$\begin{aligned} & \text{ExclusiveMergeBehaviorOfFlowNode} \Rightarrow & (A.6) \\ \Rightarrow & \text{ExclusiveMergeBehavior} : \text{FLOW\_NODES} \end{aligned}$$

$$\begin{aligned} & \text{ExclusiveSplitBehaviorOfFlowNodeInInstance} \Rightarrow & (A.7) \\ \Rightarrow & \text{ExclusiveSplitBehavior} : \text{FLOW\_NODES} \times \text{INSTANCES} \end{aligned}$$

$$\begin{aligned} & \text{InclusiveMergeBehaviorOfFlowNode} \Rightarrow & (A.8) \\ \Rightarrow & \text{InclusiveMergeBehavior} : \text{FLOW\_NODES} \end{aligned}$$

InclusiveSplitBehaviorOfFlowNodeInInstance  $\Rightarrow$  (A.9)  
 $\Rightarrow$  InclusiveSplitBehavior : FLOW\_NODES  $\times$  INSTANCES

InputBehaviorOfFlowNode  $\Rightarrow$  (A.10)  
 $\Rightarrow$  InputBehavior : FLOW\_NODES

MergeBehaviorOfFlowNode  $\Rightarrow$  (A.11)  
 $\Rightarrow$  MergeBehavior : FLOW\_NODES

ParallelMergeBehaviorOfFlowNode  $\Rightarrow$  (A.12)  
 $\Rightarrow$  ParallelMergeBehavior : FLOW\_NODES

SplitGivenFlowNodeInInstances  $\Rightarrow$  (A.13)  
 $\Rightarrow$  Split : FLOW\_NODES  $\times$  INSTANCES

SplitBehaviorOfFlowNodeAndInstance  $\Rightarrow$  (A.14)  
 $\Rightarrow$  SplitBehavior : FLOW\_NODES  $\times$  INSTANCES

allowedOutgoingSequenceFlowsFromFlowNode  $\Rightarrow$  (A.15)  
 $\Rightarrow$  allowedOutgoingSequenceFlows : FLOW\_NODES

canPassThroughFlowNodeUsingSequenceFlowInInstanceWithGateBehaviour  $\Rightarrow$  (A.16)  
 $\Rightarrow$  canPass : FLOW\_NODES  $\times$  SEQUENCE\_FLOWS  $\times$  INSTANCES  $\times$  GATE\_BEHAVIOR

instanceOfContext  $\Rightarrow$  (A.17)  
 $\Rightarrow$  instance : CONTEXTS

parentContextOfContext  $\Rightarrow$  (A.18)  
 $\Rightarrow$  parentContext : CONTEXTS

TransitionOfEvents  $\Rightarrow$  (A.19)  
 $\Rightarrow$  EventTransition : EVENTS

eventOccuredForFlowNodeInInstance  $\Rightarrow$  (A.20)  
 $\Rightarrow$  eventOccured : EVENTS  $\times$  INSTANCES

incomingSequenceFlowsOfFlowNode  $\Rightarrow$  (A.21)  
 $\Rightarrow$  incomingSequenceFlows : FLOW\_NODES

outgoingSequenceFlowsFromFlowNode  $\Rightarrow$  (A.22)  
 $\Rightarrow$  outgoingSequenceFlows : FLOW\_NODES

parentFlowNodeOfFlowNode  $\Rightarrow$  (A.23)  
 $\Rightarrow$  parentFlowNode : FLOW\_NODES

activationConditionOfFlowNode  $\Rightarrow$  (A.24)  
 $\Rightarrow$  activationCondition : FLOW\_NODES

activationConditionExpressionForComplexGateway  $\Rightarrow$  (A.25)  
 $\Rightarrow$  activationConditionExpression : COMPLEX\_GATEWAYS

defaultSequenceFlowOfGateway  $\Rightarrow$  (A.26)  
 $\Rightarrow$  defaultSequenceFlow : GATEWAYS

eventGatewayTypeOfEventBasedGateway  $\Rightarrow$  (A.27)  
 $\Rightarrow$  eventGatewayType : EVENT\_BASED\_GATEWAYS

gateConditionForSequenceFlow  $\Rightarrow$  (A.28)  
 $\Rightarrow$  gateConditionExpression : SEQUENCE\_FLOWS

gatewayDirectionOfGateway  $\Rightarrow$  (A.29)  
 $\Rightarrow$  gatewayDirection : GATEWAYS

directionOfGateway  $\Rightarrow$  (A.30)  
 $\Rightarrow$  gatewayDirection : GATEWAYS

instantiateUsingEventBasedGateway  $\Rightarrow$  (A.31)  
 $\Rightarrow$  instantiate : EVENT\_BASED\_GATEWAYS

sequenceFlowsToIgnoreDuringResetOfComplexGateway  $\Rightarrow$  (A.32)  
 $\Rightarrow$  sequenceFlowsToIgnoreDuringReset : COMPLEX\_GATEWAYS  $\times$  INSTANCES

waitingForStartOfComplexGateway  $\Rightarrow$  (A.33)  
 $\Rightarrow$  waitingForStart : COMPLEX\_GATEWAYS  $\times$  INSTANCES

activeInstancesOfActivity  $\Rightarrow$  (A.34)  
 $\Rightarrow$  activeInstances : ACTIVITIES

activeInstancesOfProcess  $\Rightarrow$  (A.35)  
 $\Rightarrow$  activeInstances : PROCESSES

firingInstancesOfFlowNode  $\Rightarrow$  (A.36)  
 $\Rightarrow$  firingInstances : FLOW\_NODES

instantiatingFlowNodeOfInstance  $\Rightarrow$  (A.37)  
 $\Rightarrow$  instantiatingFlowNode : INSTANCES

parentInstanceOfInstance  $\Rightarrow$  (A.38)  
 $\Rightarrow$  parentInstance : INSTANCES

codeOfExceptionNotification  $\Rightarrow$  (A.39)  
 $\Rightarrow$  code : EXCEPTION\_NOTIFICATIONS

contextOfNotification  $\Rightarrow$  (A.40)  
 $\Rightarrow$  context : NOTIFICATIONS

flowNodeOfNotification  $\Rightarrow$  (A.41)  
 $\Rightarrow$  flowNode : NOTIFICATIONS

nameOfSignalNotification  $\Rightarrow$  (A.42)  
 $\Rightarrow$  name : SIGNAL\_NOTIFICATIONS

occurrenceTimeOfNotification  $\Rightarrow$  (A.43)  
 $\Rightarrow$  occurrenceTime : NOTIFICATIONS

sourceFlowNodeOfSequenceFlow  $\Rightarrow$  (A.44)  
 $\Rightarrow$  sourceFlowNode : SEQUENCE\_FLOWS

sourceFlowNodeOfSequenceFlow  $\Rightarrow$  (A.45)  
 $\Rightarrow$  targetFlowNode : SEQUENCE\_FLOWS





## Appendix B

# The complete $BPMN_{GM}$ in a nutshell

### B.1 General functions and rules

The static function `sourceFlowNode` shown in signature 7.1 holds the source flow node of the given sequence flow [1, tab. 8.51].

```
static sourceFlowNode : SEQUENCE_FLOWS → FLOW_NODES  
                        (see signature 7.1)
```

The static function `targetFlowNode` shown in signature 7.2 defines the target flow node of the given sequence flow [1, tab. 8.51].

```
static targetFlowNode : SEQUENCE_FLOWS → FLOW_NODES  
                        (see signature 7.2)
```

The abstract derived function `firingInstances : FLOW_NODES → MULTISSET(INSTANCES)` shown in signature 7.7 defines a multiset of instances in which the given flow node fired. Since a flow node can fire in one step multiple times even in\* the same instance this function returns a multiset.

```
abstract derived firingInstances : FLOW_NODES →  
MULTISSET(INSTANCES)  
                                (see signature 7.7)
```

The derived function `evaluate` shown in signature 7.4 evaluates the given expression in the given instance and returns the result of the evaluation. The own evaluation depends on the implementation in the WFI and the `EXPRESSION` language has to be supported by the WFI.

```
abstract derived evaluate : EXPRESSIONS × INSTANCES →
  BOOLEAN
(see signature 7.4)
```

The rule `WorkflowTransition` shown in listing 7.4 handles both, the transition between the static flow nodes in the process definition and the transition between the life-cycle states of all instances of the given flow node [44].

```
rule WorkflowTransition : FLOW_NODES [B.1]
```

```
1 rule WorkflowTransition(node) =
2   parblock
3     FlowNodeTransition(node)
4     InstanceTransition(node)
5 endparblock
```

## B.2 Token related functions and rules

The controlled function `instanceOfToken` defines the activity/process instance the given token belongs to.

```
controlled instance : TOKENS → INSTANCES
(see signature 7.3)
```

The derived function `instance` : `SET(TOKENS) → INSTANCES` (see listing 7.3 receives a set of tokens and returns the instance of those tokens. It assumes that all tokens are from the same instance as defined in `enablingTokenSets` : `FLOW_NODES → SET(SET(TOKENS))` shown in listing 7.2.

```
derived instance : SET(TOKENS) → INSTANCES [B.2]
```

```
1 derived instance(tokens) =
2   return res in
3     choose t ∈ tokens do
4       res ← instance(t)
```

The derived function  $\text{enablingTokenSets} : \text{FLOW\_NODES} \rightarrow \text{SET}(\text{SET}(\text{-TOKENS}))$  shown in listing 7.2 returns set of sets of tokens. In every token set all tokens must belong to the same instance. Also a token set must not contain two tokens residing in the same sequence flow. For example, if there will be a flow node with three incoming sequence flows  $SF_1$ ,  $SF_2$  and  $SF_3$ , on which tokens from two different instances,  $I_A$  and  $I_B$ , will reside and the token distribution will be the following: sequence flow  $SF_1$  contains two tokens of instance  $I_A$  and one token of instance  $I_B$ , sequence flow  $SF_2$  contains one token of each instance and sequence flow  $SF_3$  contains two tokens of instance  $I_B$  than the following four token sets will be returned: Set 1 will contain two tokens of instance  $I_A$  from sequence flow  $SF_1$  and  $SF_2$ . Set 2 will contain one token of instance  $I_A$  from sequence flow  $SF_1$ . Set 3 will contain three tokens of instance  $I_B$  from all the three incoming sequence flows. And last, set 4 will contain one token of instance  $I_B$  from the sequence flow  $SF_3$ . Each of the token sets may or may not fire the given flow node depending on the  $\text{MergeBehavior} : \text{FLOW\_NODES}$  (see signature 7.8).

derived  $\text{enablingTokenSets} : \text{FLOW\_NODES} \rightarrow \text{SET}(\text{SET}(\text{-TOKENS}))$  [B.3]

```

1 derived enablingTokenSets(flowNode) =
2   return res in
3     local selectedToken, allTokens, tokenSet  $\leftarrow \emptyset$  in
4       res  $\leftarrow \emptyset$ 
5       allTokens  $\leftarrow \{ \text{token} \mid \text{token} \in \text{TOKENS} \}$ 
6          $\wedge \text{sequenceFlow}(\text{token}) \in \text{incomingSequenceFlows}(\text{flowNode}) \}$ 
7
8     while allTokens  $\neq \emptyset$  do
9       selectedToken  $\leftarrow \text{undef}$ 
10
11     if tokenSet =  $\emptyset$  then
12       choose token  $\in$  allTokens do selectedToken  $\leftarrow$  token
13     else
14       choose token  $\in$  allTokens :
15          $\forall t \in \text{tokenSet}$  with
16           sequenceFlow(t)  $\neq$  sequenceFlow(token)
17            $\wedge \text{instance}(t) = \text{instance}(\text{token})$  holds do
18             selectedToken  $\leftarrow$  token
19
20     if selectedToken  $\neq \text{undef}$  then
21       add selectedToken to tokenSet
22       remove selectedToken from allTokens
23
24     if selectedToken =  $\text{undef} \vee \text{allTokens} = \emptyset$  then
25       add tokenSet to res
26       tokenSet  $\leftarrow \emptyset$ 

```

The  $\text{ColorProcessGraph}$  rule shown in 8.11 finds a shortest path from the root node of  $\mathcal{G}_k$  (representing an inclusive or complex gateway in  $\mathcal{G}_d$ ) given as the only parameter to any reachable node in the directed graph  $\mathcal{G}_k$  in the way the original Dijkstra's algorithm [59] was designed. Additionally the algorithm colors visited

nodes and tested edges, even those tested edges which are not further used for the search (i.e., incoming edges of closed nodes  $n \in \mathcal{N}$  indicated by  $\kappa(\text{root}, n) = \top$ ). The algorithm starts with coloring each of the directly outgoing edges from the root node with a distinct color. Those colors are then populated through  $\mathcal{G}_k$  till its leafs (usually representing start events in  $\mathcal{G}_d$ ). If the algorithm visits a node which has more than one incoming edge in  $\mathcal{G}_k$  (representing a splitting gateway in  $\mathcal{G}_d$ ) for the first time (i.e., such node  $n \in \mathcal{N}$  is not closed yet:  $\kappa(\text{root}, n) = \perp$ ), it will additionally color this node with a distinct limpoid color. Such limpoid color will be further populated with the other colors such a node is colored with. A color calculation for a root node (represented by a run of ColorProcessGraph) will never color two nodes in  $\mathcal{G}_k$  having more than one incoming edge with the same limpoid color. A limpoid color is an indicator for other alternative paths of the concrete node to populate their own non-limpoid colors. Every time such a node will be tested by the algorithm again (i.e., such a node is already closed), all non-limpoid colors, of the tested alternative edge will be populated to all nodes and edges in  $\mathcal{G}_k$  which are colored with the limpoid color of the tested node.

All limpoid colors are only visible locally inside the run of the ColorProcessGraph. The resulting set of colors obtained from both color functions ( $\mathcal{C}_{\mathcal{N}}, \mathcal{C}_{\mathcal{E}}$ ) will not contain any limpoid colors.

rule ColorProcessGraph : N [B.4]

```

1 rule ColorProcessGraph(root) =
2    $\forall n \in \mathcal{N}$  do
3      $\kappa(\text{root}, n) \leftarrow \perp$ 
4      $\delta(\text{root}, n) \leftarrow \infty$ 
5      $\mathcal{C}_{\mathcal{N}}(\text{root}, n) \leftarrow \emptyset$ 
6
7    $\forall e \in \mathcal{E}$  do  $\mathcal{C}_{\mathcal{E}}(\text{root}, e) \leftarrow \emptyset$ 
8    $\delta(\text{root}, \text{root}) \leftarrow 0$ 
9   local set  $\leftarrow \{\text{root}\}$ , color in
10    while  $|\text{set}| > 0$  do
11      choose node  $\in \{n \mid n \in \text{set}\}$ 
12       $\wedge \nexists m \in \text{set} : \delta(\text{root}, n) > \delta(\text{root}, m)$  holds }do
13
14      remove node from set
15       $\kappa(\text{root}, \text{node}) \leftarrow \top$ 
16
17      foreach next  $\in \mathcal{N}$  with  $\mathcal{E}(\text{node}, \text{next}) \neq \text{undef} \wedge \text{next} \neq \text{root}$  do
18        if  $\delta(\text{root}, \text{node}) + 1 < \delta(\text{root}, \text{next})$ 
19           $\wedge \neg \kappa(\text{root}, \text{next})$  then
20
21           $\delta(\text{root}, \text{next}) \leftarrow \delta(\text{root}, \text{node}) + 1$ 
22          add next to set
23
24          if  $\exists n \in \mathcal{N}$  with  $\mathcal{E}(n, \text{next}) \neq \text{undef} \wedge n \neq \text{node}$  then
25            add nextLimpoid to  $\mathcal{C}_{\mathcal{N}}(\text{root}, \text{node})$ 
26
27      if node = root then
28        color  $\leftarrow \text{nextColor}$ 

```

```

29      add color to  $\mathcal{C}_{\mathcal{N}}(\text{root}, \text{next})$ 
30      add color to  $\mathcal{C}_{\mathcal{E}}(\text{root}, \mathcal{E}(\text{node}, \text{next}))$ 
31    else
32      foreach color  $\in \mathcal{C}_{\mathcal{N}}(\text{root}, \text{node})$  do
33        if ( $\nexists c \in \mathcal{C}_{\mathcal{N}}(\text{root}, \text{next}) : \text{isLimpid}(c)$  holds)
34           $\vee \neg \text{isLimpid}(\text{color})$  then
35            add color to  $\mathcal{C}_{\mathcal{N}}(\text{root}, \text{next})$ 
36
37      add color to  $\mathcal{C}_{\mathcal{E}}(\text{root}, \mathcal{E}(\text{node}, \text{next}))$ 
38
39      foreach limpid  $\in \mathcal{C}_{\mathcal{N}}(\text{root}, \text{next})$  with isLimpid(limpid) do
40         $\forall n \in \mathcal{N}$  with limpid  $\in \mathcal{C}_{\mathcal{N}}(\text{root}, n)$  do
41          add color to  $\mathcal{C}_{\mathcal{N}}(\text{root}, n)$ 
42
43         $\forall e \in \mathcal{E}$  with limpid  $\in \mathcal{C}_{\mathcal{E}}(\text{root}, e)$  do
44          add color to  $\mathcal{C}_{\mathcal{E}}(\text{root}, e)$ 

```

An upstream token of a flow node, defined in listing 8.12, is a token in a sequence flow if there is a path starting with that sequence flow and reaching the given flow node with all other sequence flows building that path not having a token, if such a sequence flow is not directly connected to the given flow node and there is no alternative path from that sequence flow to the given flow node reaching a directly connected sequence flow to the given flow node having a token. Upstream tokens are needed for flow nodes such as inclusive and complex gateways. This is defined using inhibiting and anti-inhibiting paths [60] as a token, which has an inhibiting path but no anti-inhibiting path to the corresponding flow node. Such upstream tokens are used as an activation condition of an inclusive gateway [1, tab. 13.3], or as a reset condition of a complex gateway [1, tab. 13.5].

```

rule UpstreamTokens : FLOW_NODES  $\times$  SET(SEQUENCE_FLOWS)  $\times$ 
SET(TOKENS)  $\rightarrow$  SET(TOKENS) [B.5]

```

```

1 rule UpstreamTokens(flowNode, sequenceFlows, tokens) =
2   // colors to ignore
3   let ignore  $\leftarrow \{ c \mid \exists e \in \text{sequenceFlows} \text{ with}$ 
4     |tokensInSequenceFlow(e)  $\cap$  tokens|  $> 0$  holds
5      $\wedge c \in \mathcal{C}_{\mathcal{E}}(\text{flowNode}, e) \}$  in
6   // all relevant sequence flows in the diagram
7   let relevant  $\leftarrow \{ e \mid e \in \text{EDGES}$ 
8      $\wedge \mathcal{C}_{\mathcal{E}}(\text{flowNode}, e) \setminus \text{ignore} \neq \emptyset \}$  in
9
10  return res in
11    res  $\leftarrow \{ t \mid \exists e \in \text{relevant}$ 
12       $\wedge \exists t \in \text{tokensInSequenceFlow}(e) \text{ with}$ 
13        instance(t) = instance(tokens)
14         $\wedge (\mathcal{C}_{\mathcal{E}}(\text{flowNode}, e) \setminus \text{ignore}) \neq \emptyset \text{ holds } \}$ 

```

## B.3 Behaviors

### B.3.1 Gate behavior

Derived function `allowedOutgoingSequenceFlows : FLOW_NODES → SET(SEQUENCE_FLOWS)` shown in signature 7.5 is an abstract function used in `SplitBehavior` (see signature 7.9) to determine those outgoing sequence flows which will get a token. This abstract function has to be further refined in the concrete behaviors such as `ExclusiveSplitBehavior` (see listing 7.21), `InclusiveSplitBehavior` (see listing 7.22), and `ParallelSplitBehavior` (see listing 7.20).

```
abstract derived allowedOutgoingSequenceFlows : FLOW-  
_NODES → SET(SEQUENCE_FLOWS)  
  
(see signature 7.5)
```

The derived function `canPass` shown in listing 7.5 defines the conditional flow possibilities for a BPMN diagram. The first parameter is the flow node the gate behavior is defined for, the second parameter holds the concrete outgoing sequence flow of that flow node, the third parameter identifies the instance of the flow node, and the forth parameter defines if the behavior should be "EXCLUSIVE", "INCLUSIVE", or "PARALLEL".

```
derived canPass : FLOW_NODES × SEQUENCE_FLOWS ×  
INSTANCES × GATE_BEHAVIOR → BOOLEAN  
  
[B.6]
```

```
1 derived canPass(flowNode, sequenceFlow, instance, behavior) =  
2   return res in  
3     let others ← outgoingSequenceFlows(flowNode) \ sequenceFlow in  
4       if (behavior = "EXCLUSIVE" ∨ behavior = "INCLUSIVE")  
5         ∧ gateConditionExpression(sequenceFlow) = "DEFAULT" then  
6         res ← ∀ other ∈ others :  
7           ¬ evaluate(gateConditionExpression(other), instance) holds  
8  
9       else if behavior = "EXCLUSIVE" then  
10        res ← evaluate(gateConditionExpression(sequenceFlow), instance) = T  
11        ∧ ∀ otherSequenceFlow ∈ { sf | sf ∈ others  
12          ∧ gateConditionExpression(sf) ≠ "DEFAULT" } :  
13          ¬ evaluate(gateConditionExpression(otherSequenceFlow),  
14            instance) holds  
15  
16       else if behavior = "INCLUSIVE" then  
17        res ← evaluate(gateConditionExpression(sequenceFlow), instance)  
18  
19       else if behavior = "PARALLEL" then  
20        res ← T
```

### B.3.2 Activation behavior

The `InputBehavior` shown in listing 7.6 defines the input behavior for of a flow node. This rule returns all relevant instances in which the flow node should fire. The size of the multiset identifies the number of fires. The tokens, which contributed to any of the fires should be consumed (see listing 7.7).

This rule distinguishes between two basic cases. First is the case when only one incoming sequence flow is targeting the given flow node. Second case is when multiple sequence flows are targeting the given flow node. This may be seen as unnecessary and we explicitly distinguish between those case to relate to the BPMN [1] specification, which in some places makes this separation explicitly, e.g., in the case of `gatewayDirection` attribute in gateways. On a closer look it can be observed that all the merging behaviors would in case of only one incoming sequence flow behave in the same way as the `SoleInputBehavior` does. The only difference is that the `SoleInputBehavior` uses the `OneIncomingSequenceFlow` constraint restricting the number of incoming sequence flow targeting the given flow node to the cardinality one. This allows us to define flow nodes where only one incoming sequence flow would be a valid configuration by defining the `Merge Behavior` as `SoleInputBehavior`.

```
rule InputBehavior : FLOW_NODES → MULTISSET(-  
INSTANCES) [B.7]
```

```
1 rule InputBehavior(flowNode) =  
2   return res in
```

Will chose merging behavior only for multiple incoming sequence flow

```
4   if |incomingSequenceFlows(flowNode)| = 1 then  
5     res ← SoleInputBehavior(flowNode)  
6   else  
7     res ← MergeBehavior(flowNode)
```

The rule `ActivationBehavior : FLOW_NODES → MULTISSET(-INSTANCES)` shown in listing 7.7 defines the abstract behavior for activation of a flow node. This rule returns all relevant instances in which the flow node fired. The size of the returned multiset identifies the number of fires. The tokens, which contributed to any of the fires are returned by the abstract derived function `firingTokens : FLOW_NODES → SET(TOKENS)` (see signature 7.6) and consumed here. The instances in which the given flow node fired are returned using the abstract derived function `firingInstances : FLOW_NODES → MULTISSET(-INSTANCES)` (see signature 7.7). Those two abstract derived functions need to be further refined in subrules defining the concrete sole or merging behavior.

```
rule ActivationBehavior : FLOW_NODES → MULTISSET(-  
INSTANCES) [B.8]
```

```

1 rule ActivationBehavior(flowNode) =
2   return res in
3      $\forall$  token  $\in$  firingTokens(flowNode) do
4       ConsumeToken(token)
5
6   res  $\leftarrow$  firingInstances(flowNode)

```

The sole input behavior shown in listing 7.8 refines the ActivationBehavior : FLOW\_NODES  $\rightarrow$  MULTISSET(INSTANCES) rule (see listing 7.7) for the usage in flow nodes, which have, or are allowed to have, only one incoming sequence flow. In this case every token set in enablingTokenSets will contain exactly one token. Consequently, all tokens in all enablingTokenSets are firingTokens and every instance of every such firing token is a firing instance. Therefore, the function firingInstances holds instances of all tokens residing on the one incoming sequence flow.

```

rule SoleInputBehavior : FLOW_NODES  $\rightarrow$  MULTISSET(-
INSTANCES)

```

[B.9]

```

1 rule SoleInputBehavior(flowNode) =
2   assert OneIncomingSequenceFlow(flowNode)
3
4   return res in
5     let ft  $\leftarrow$   $\bigcup$  enablingTokenSets(flowNode) in
6     res  $\leftarrow$  ActivationBehavior(flowNode) where
7       firingTokens(flowNode) = ft
8       firingInstances(flowNode) = [ i |  $\exists$  t  $\in$  ft  $\wedge$  instance(t) = i ]

```

Will raise an error with the name of the constraint if the given flow node has more than one incoming sequence flows.

```

constraint OneIncomingSequenceFlow : FLOW_NODES

```

[B.10]

```

1 constraint OneIncomingSequenceFlow(flowNode) =
2   | incomingSequenceFlows(flowNode) | = 1

```

The MergeBehavior shown in signature 7.8 defines the abstract behavior pattern for merging multiple sequence flows of a flow node. This rule returns all relevant instances of a (sub-)process containing the given flow node in which the flow node fired.

```

abstract rule MergeBehavior : FLOW_NODES

```

(see signature 7.8)



The rule `ParallelMergeBehavior : FLOW_NODES → MULTISSET(-INSTANCES)` shown in listing 7.10 refines the rule `ActivationBehavior : FLOW_NODES → MULTISSET(INSTANCES)` (see signature B.8), for the usage in parallel gateways. This behavior defines `firingTokenSets` as enabling-TokenSets, which have tokens in all incoming sequence flows of the given flow node. The `firingTokens` are then all tokens in all such `firingTokenSets` and `firingInstances` are instances per firing token set (see listing 7.3).

```
rule ParallelMergeBehavior : FLOW_NODES → MULTISSET(-
INSTANCES) [B.11]
```

```
1 rule ParallelMergeBehavior(flowNode) =
2   return res in
3     let firingTokenSets ← { ts | ts ∈ enablingTokenSets(flowNode)
4       ∧ ∀ sf ∈ incomingSequenceFlows(flowNode) : (
5         ∃ t ∈ ts with
6           t ∈ tokensInSequenceFlow(sf)) holds } in
7     res ← ActivationBehavior(flowNode) where
8       firingTokens(flowNode) = ∪ firingTokenSets
9       firingInstances(flowNode) = [ i | ∃ ts ∈ firingTokenSets
10        ∧ instance(ts) = i ]
```

The rule `ExclusiveMergeBehavior : FLOW_NODES → MULTISSET(-INSTANCES)` shown in listing 7.11 refines the rule `ActivationBehavior : FLOW_NODES → MULTISSET(INSTANCES)` (see signature B.8) for the usage in, e.g., exclusive gateways or activities. This behavior fires the given flow node if at least one token is present on any of the incoming sequence flows. No matter if only one or more tokens are present on the incoming sequence flows all will be consumed and for each of them the given flow node will fire. In other words it also means that this behavior will fire the given flow node for each token on any of the incoming sequence flows in any instance and may also fire multiple times in the same instance (see signature 7.7).

```
rule ExclusiveMergeBehavior : FLOW_NODES → MULTISSET(-
INSTANCES) [B.12]
```

```
1 rule ExclusiveMergeBehavior(flowNode) =
2   return res in
3     let ft ← ∪ enablingTokenSets(flowNode) in
4     res ← ActivationBehavior(flowNode) where
5       firingTokens(flowNode) = ft
6       firingInstances(flowNode) = [ i | ∃ t ∈ ft ∧ instance(t) = i ]
```

If a gateway is allowed to have only one incoming sequence flow containing a token the rule `MutuallyExclusiveMergeBehavior` shown in listing 7.12 should

be used. Thus, more than one incoming sequence flow containing a token is considered as a violation of this rule and will raise an `MultipleTokensAtMutuallyExclusiveMerge` error. On the other hand multiple tokens in the same incoming sequence flow will not raise such an error. This is considered as correct behavior and will fire the given flow node for each token residing in such a sequence flow.

```
rule MutuallyExclusiveMergeBehavior : FLOW_NODES →
MULTISET(INSTANCES) [B.13]
```

```
1 rule MutuallyExclusiveMergeBehavior(flowNode) =
2   assert MultipleTokensAtMutuallyExclusiveMerge(flowNode)
3
4   return res in
5     res ← ExclusiveMergeBehavior(flowNode)
```

The `MultipleTokensAtMutuallyExclusiveMerge` shown in listing 7.13 holds if only one incoming sequence flows has a token. More than one token on the same sequence flow is possible.

```
constraint MultipleTokensAtMutuallyExclusiveMerge : FLOW-
_NODES [B.14]
```

```
1 constraint MultipleTokensAtMutuallyExclusiveMerge(flowNode) =
2   ∀ tokenSet ∈ enablingTokenSets(flowNode) : |tokenSet| ≤ 1 holds
```

The rule `InclusiveMergeBehavior : FLOW_NODES → MULTISET(-INSTANCES)` shown in listing 7.14 refines the rule `ActivationBehavior : FLOW_NODES → MULTISET(INSTANCES)` and is used, e.g., in inclusive gateways. This behavior fires the given flow node if at least one token is present on any of the incoming sequence flows and no token in the process “may still arrive” [60]. This is defined as upstream token and the presence of such upstream token is checked using the `UpstreamTokens : FLOW_NODES × SET(SEQUENCE_FLOWS) × SET(TOKENS) → SET(TOKENS)` rule. If no such upstream token exists, this behavior will fire for every firing token set which meets the above condition.

```
rule InclusiveMergeBehavior : FLOW_NODES → MULTISET(-
INSTANCES) [B.15]
```

```
1 rule InclusiveMergeBehavior(flowNode) =
2   let firingTokenSets ← { ts |
3     UpstreamTokens(flowNode, incomingSequenceFlows(flowNode), ts) = 0 } in
4   ActivationBehavior(flowNode) where
5     firingTokens(flowNode) = ⋃ firingTokenSets
6     firingInstances(flowNode) = [ i | ∃ ts ∈ firingTokenSets
7       ∧ instance(ts) = i ]
```

The rule `ComplexMergeBehavior : FLOW_NODES → MULTISSET(-INSTANCES)` shown in listing 7.15 takes, compared to other merge behaviors, an internal state of the complex gateway into account [1]. This state is also handled only in this rule. It may be read for the purpose of conditions in outgoing sequence flows but not written. Firing a flow node using this `ComplexMergeBehavior` in `waitingForStart` conditioned only by the `activationConditionExpression : COMPLEX_GATEWAYS → EXPRESSIONS`. In the “waiting for reset” state [1] the firing condition is similar to the `InclusiveMergeBehavior : FLOW_NODES → MULTISSET(INSTANCES)` shown in listing 7.14. The difference is that the relevant sequence flows, where the flow node may wait for a token, which “may still arrive”, does not include the sequence flows which contributed to the activation in the first phase [1].

The BPMN 2.0 specification does not say anything about the quantity of tokens and the `activationConditionExpression` does not define any internal structure from which it would be possible to obtain the concrete tokens to be consumed. Therefore, we cannot use our `enablingTokenSets` in the way as other merge behaviors are using it and we define the `ComplexMergeBehavior` as follows (see listing 7.15: A set of possible firing instances ( $fi_P$ ) in which the flow node may fire is defined as a set of instances where there is a enabling token set for the given flow node in any of such instances. Then the firing instances in `waitingForStart` state ( $fi_S$ ) are all possible firing instances ( $fi_P$ ) in which the flow node is in `waitingForStart` state and the `activationConditionExpression` evaluates in such instances to true. The possible firing instances in “waiting for reset” state ( $fi_{RP}$ ) are all possible firing instances ( $fi_P$ ) in which the flow node is in not `waitingForStart` state. An important property of the firing instances in `waitingForStart` state ( $fi_S$ ) and the possible firing instances in “waiting for reset” state ( $fi_{RP}$ ) is that they are distinct sets due to the `waitingForStart` condition, i.e.,  $fi_S \cap fi_{RP} = \emptyset$ .

Next, we need to extract firing token sets from `enablingTokenSets` for each state. The first, firing token sets in `waitingForStart` state ( $fts_S$ ) are all `enablingTokenSets` which are in one of the firing instances in `waitingForStart` state ( $fi_S$ ). The second, firing token sets in “waiting for reset” state ( $fts_R$ ) which are in one of the possible firing instances in “waiting for reset” state ( $fi_{RP}$ ) and there is a token in such enabling token set which does not reside in one of the `sequenceFlowsToIgnoreDuringReset`. Since the firing instances in `waitingForStart` state ( $fi_S$ ) and the possible firing instances in “waiting for reset” state ( $fi_{RP}$ ) are two distinct sets and that all tokens in one enabling token set belong to the same instance (see listing 7.2), the firing token sets in `waitingForStart` state ( $fts_S$ ) and the firing token sets in “waiting for reset” state ( $fts_R$ ) are also distinct sets, i.e.,  $fts_S \cap fts_R = \emptyset$ .

Now we can define the firing instances in “waiting for reset” state ( $fi_R$ ) as possible firing instances in “waiting for reset” state ( $fi_{RP}$ ) where there is no upstream token for the given flow node in relevant incoming sequence flows with relevant tokens residing on those relevant incoming sequence flows. Relevant incoming sequence flows are incoming sequence flows to the given flow node without `sequenceFlowsToIgnoreDuringReset`.

Next, all sequence flows which contributed to the activation in the `waiting-`

ForStart state will be marked as `sequenceFlowsToIgnoreDuringReset` and the `waitingForStart` state will be set for all firing instances in `waitingForStart` state ( $fi_S$ ) to false and for all firing instances in “waiting for reset” state ( $fi_R$ ) to true.

Last, the `ActivationBehavior` (see listing 7.7) will be refined by defining the `firingTokens` as a union of all tokens from all firing token sets in `waitingForStart` state ( $fts_S$ ) and all tokens from firing token sets in “waiting for reset” state ( $fts_R$ ) which do not reside on `sequenceFlowsToIgnoreDuringReset`. The `firingInstances` of the `ActivationBehavior` are defined as a union of all firing instances in `waitingForStart` state ( $fi_S$ ) and all firing instances in “waiting for reset” state ( $fi_R$ ).

```
rule ComplexMergeBehavior : FLOW_NODES → MULTISSET(-
INSTANCES) [B.16]
```

```

1 rule ComplexMergeBehavior(flowNode) =
2   return res in
3     let  $fi_P \leftarrow \{i \mid \exists ts \in \text{enablingTokenSets}(\text{flowNode}) \wedge \text{instance}(ts) = i\}$  in
4     let  $fi_S \leftarrow \{i \mid \exists i \in fi_P : \text{waitingForStart}(\text{flowNode}, i) = \top$ 
5        $\wedge \text{evaluate}(\text{activationConditionExpression}(\text{flowNode}), i) = \top \text{ holds}$ 
6        $\}$ ,
7     let  $fi_{RP} \leftarrow \{i \mid \exists i \in fi_P : \text{waitingForStart}(\text{flowNode}, i) = \perp\}$  in
8     let  $fts_S \leftarrow \{ts \mid$ 
9        $ts \in \text{enablingTokenSets}(\text{flowNode}) \wedge \text{instance}(ts) \in fi_S\}$ ,
10    let  $fts_R \leftarrow \{ts \mid ts \in \text{enablingTokenSets}(\text{flowNode}) \wedge \text{instance}(ts) \in fi_{RP}$ 
11       $\wedge \exists t \in ts : \text{sequenceFlow}(t) \notin \text{sequenceFlowsToIgnoreDuringReset}(\text{flowNode}, \text{instance}(ts)) \text{ holds } \}$  in
12    let  $fi_R \leftarrow \{i \mid i \in fi_{RP}$ 
13       $\wedge \text{UpstreamTokens}(\text{flowNode},$ 
14         $\text{incomingSequenceFlows}(\text{flowNode}) \setminus$ 
15         $\text{sequenceFlowsToIgnoreDuringReset}(\text{flowNode}, i),$ 
16         $\{t \mid t \in \text{UNION } fts_R$ 
17         $\wedge t \notin \text{sequenceFlowsToIgnoreDuringReset}(\text{flowNode}, i)\} =$ 
18         $\emptyset \}$  in
19
20     $\forall i \in fi_S \text{ do}$ 
21       $\text{sequenceFlowsToIgnoreDuringReset}(\text{flowNode}, i) \leftarrow \{sf \mid$ 
22         $sf \in \text{incomingSequenceFlows}(\text{flowNode})$ 
23         $\wedge \exists ts \in fts_S$ 
24         $\wedge \exists t \in ts : \text{sequenceFlow}(t) = sf \text{ holds } \}$ 
25       $\text{waitingForStart}(\text{flowNode}, i) \leftarrow \perp$ 
26
27     $\forall i \in fi_R \text{ do } \text{waitingForStart}(\text{flowNode}, i) \leftarrow \top$ 
28
29    res  $\leftarrow \text{ActivationBehavior}(\text{flowNode}) \text{ where}$ 
30       $\text{firingTokens}(\text{flowNode}) = (\text{UNION } fts_S) \cup \{t \mid t \in \text{UNION } fts_R$ 
31         $\wedge t \notin \text{sequenceFlowsToIgnoreDuringReset}(\text{flowNode}, i)\}$ 
32       $\text{firingInstances}(\text{flowNode}) = \bigcup (fi_S \cup fi_R)$ 
```

### B.3.3 Output behavior

The rule `OutputBehavior : FLOW_NODES × MULTISSET(INSTANCES)` shown in listing 7.16 defines the abstract behavior for producing tokens on outgoing sequence flows of a flow node.

```
rule OutputBehavior : FLOW_NODES × MULTISSET(-  
INSTANCES) [B.17]
```

```
1 rule OutputBehavior(flowNode, instances) =  
2   ∀ instance ∈ instances do  
3     if |outgoingSequenceFlows(flowNode)| = 1 then  
4       SoleOutputBehavior(flowNode, instance)  
5     else  
6       SplitBehavior(flowNode, instance)
```

The rule `SoleOutputBehavior : FLOW_NODES × INSTANCES` shown in listing 7.17 defines the behavior for producing a token for flow nodes with exactly one outgoing sequence flow. This is realized with the `OneOutgoingSequenceFlow` constraint.

```
rule SoleOutputBehavior : FLOW_NODES × INSTANCES [B.18]
```

```
1 rule SoleOutputBehavior(flowNode, instance) =  
2   assert OneOutgoingSequenceFlow(flowNode)  
3  
4   choose sequenceFlow ∈ outgoingSequenceFlows(flowNode) do  
5     Split(flowNode, instance) where  
6       allowedOutgoingSequenceFlows(flowNode) = { sequenceFlow }
```

Will raise an error with the name of the constraint if the given flow node has more than one outgoing sequence flows.

```
constraint OneOutgoingSequenceFlow : FLOW_NODES [B.19]
```

```
1 constraint OneOutgoingSequenceFlow(flowNode) =  
2   |outgoingSequenceFlows(flowNode)| = 1
```

Abstract split behavior rule shown in signature 7.9 is used in other behaviors where it is refined to implement a concrete split behavior.

```
abstract rule SplitBehavior : FLOW_NODES × INSTANCES  
                                (see signature 7.9)
```

The rule `Split : FLOW_NODES × INSTANCES` shown in listing 7.19 defines the splitting of the control flow originating in the given flow node into parallel or alternative paths. This is done by producing a token on all `allowedOutgoingSequenceFlows` of the given flow node.

rule `Split : FLOW_NODES × INSTANCES` [B.20]

```

1 rule Split(flowNode, instance) =
2   ∀ sequenceFlow ∈ allowedOutgoingSequenceFlows(flowNode) do
3     ProduceToken(sequenceFlow, instance)

```

The `ParallelSplitBehavior : FLOW_NODES × INSTANCES` shown in listing 7.20 refines the rule `SplitBehavior : FLOW_NODES × INSTANCES`. This behavior is used as a default splitting behavior in BPMN 2.0 for any flow node except exclusive, inclusive, complex and event-based gateways. This behavior defines all outgoing sequence flows as allowed and will produce a token on all of them (see: listing 7.5).

rule `ParallelSplitBehavior : FLOW_NODES × INSTANCES` [B.21]

```

1 rule ParallelSplitBehavior(flowNode, instance) =
2   Split(flowNode, instance) where
3     allowedOutgoingSequenceFlows(flowNode) =
4     { sequenceFlow | sequenceFlow ∈ outgoingSequenceFlows(flowNode)
5       ∧ canPass(flowNode, sequenceFlow, instance, "PARALLEL") }

```

The `ExclusiveSplitBehavior : FLOW_NODES × INSTANCES` shown in listing 7.21 refines the `SplitBehavior : FLOW_NODES × INSTANCES`, which is used in exclusive gateways. This behavior allows to produce a token on exactly one outgoing sequence flow as defined in `canPass`.

rule `ExclusiveSplitBehavior : FLOW_NODES × INSTANCES` [B.22]

```

1 rule ExclusiveSplitBehavior(flowNode, instance) =
2   Split(flowNode, instance) where
3     allowedOutgoingSequenceFlows(flowNode) =
4     { sequenceFlow | sequenceFlow ∈ outgoingSequenceFlows(flowNode)
5       ∧ canPass(flowNode, sequenceFlow, instance, "EXCLUSIVE") }

```

The `InclusiveSplitBehavior : FLOW_NODES × INSTANCES` shown in listing 7.22 refines the rule `SplitBehavior : FLOW_NODES × INSTANCES`, which is used in inclusive and complex gateways. It allows to produce a token on any subset of the outgoing sequence flows (see: listing 7.5).

```
rule InclusiveSplitBehavior : FLOW_NODES × INSTANCES [B.23]
```

```

1 rule InclusiveSplitBehavior(flowNode, instance) =
2   Split(flowNode, instance) where
3     allowedOutgoingSequenceFlows(flowNode) =
4       { sequenceFlow | sequenceFlow ∈ outgoingSequenceFlows(flowNode)
5         ∧ canPass(flowNode, sequenceFlow, instance, "INCLUSIVE") }

```

## B.4 Flow nodes

The derived function `incomingSequenceFlows` shown in listing 7.23 returns the set of all incoming sequence flows for the given flow node.

```
derived incomingSequenceFlows : FLOW_NODES → SET(-
SEQUENCE_FLOWS) [B.24]
```

```

1 derived incomingSequenceFlows(flowNode) =
2   { x | x ∈ SEQUENCE_FLOWS ∧ targetFlowNode(x) = flowNode }

```

The derived function `outgoingSequenceFlows` shown in listing 7.24 returns the set of all outgoing sequence flows for the given flow node.

```
derived outgoingSequenceFlows : FLOW_NODES → SET(-
SEQUENCE_FLOWS) [B.25]
```

```

1 derived outgoingSequenceFlows(flowNode) =
2   { x | x ∈ SEQUENCE_FLOWS ∧ sourceFlowNode(x) = flowNode }

```

The static function `parentFlowNode` shown in signature 7.10 defines the parent node of the given flow node. Since only activities can have child flow nodes (e.g. sub-process) this function returns only items from the universe `ACTIVITIES`.

```
static parentFlowNode : FLOW_NODES → ACTIVITIES
                        (see signature 7.10)
```

The rule `FlowNodeTransition : FLOW_NODES` checks required conditions for a flow node regarding events, control flow, data, and resources, and performs the corresponding operations. This rule represents the most abstract transition rule for flow nodes and is further refined on lower levels for activities, events, and gateways, and further for specific types of these flow nodes.

<code>rule FlowNodeTransition : FLOW_NODES</code>	[B.26]
---	--------

```

1 rule FlowNodeTransition(flowNode) =
2   if activationCondition(flowNode)
3      $\wedge$  dataCondition(flowNode)
4      $\wedge$  resourceCondition(flowNode)
5   then parblock
6     FlowNodeOperation(flowNode)
7     DataOperation(flowNode)
8     ResourceOperation(flowNode)
9   endparblock

```

The `activationCondition` shown in signature 7.11 defines activation conditions of the given flow node. E.g., if there are enough tokens on the incoming sequence flows of the given flow node or if a certain event occurred.

<code>abstract derived activationCondition : FLOW_NODES <math>\rightarrow</math> BOOLEAN</code>	(see signature 7.11)
---	----------------------

The `dataCondition` shown in signature 7.12 determines if data constraints are met before activating the given flow node. It is currently not specified for any transition in the current ground model and has to be refined as soon as the data aspect is included.

<code>abstract derived dataCondition : FLOW_NODES <math>\rightarrow</math> BOOLEAN</code>	(see signature 7.12)
---	----------------------

The `resourceCondition` shown in signature 7.13 determines if resource constraints are met before activating the given flow node. Currently it is kept abstract for all transitions because the BPMN standard does not sufficiently specify resources.

<code>abstract derived resourceCondition : FLOW_NODES <math>\rightarrow</math> BOOLEAN</code>	(see signature 7.13)
---	----------------------

The abstract rule `FlowNodeOperation` shown in signature 7.14 is responsible for flow node related work, e.g., producing tokens on the outgoing sequence flows, throwing an event, creating instances of tasks and sub-processes. It has to be refined for all transitions of non-abstract meta-model classes.



```
abstract rule FlowNodeOperation : FLOW_NODES
                                         (see signature 7.14)
```

The abstract rule `DataOperation` shown in signature 7.15 is responsible for data. It is currently not specified for any transition in the current ground model, and has to be refined as soon as the data aspect is included.

```
abstract rule DataOperation : FLOW_NODES
                                         (see signature 7.15)
```

The abstract rule `ResourceOperation` shown in signature 7.16 is responsible for resources. Currently it is kept abstract for all transitions because the BPMN standard does not sufficiently specify resources.

```
abstract rule ResourceOperation : FLOW_NODES
                                         (see signature 7.16)
```

## B.5 Activities

The monitored function `completed` shown in signature 7.17 indicates if the given activity in a given instance has been completed. This location must be set, e.g., for tasks when their work is finished.

```
shared completed : ACTIVITIES × INSTANCES → BOOLEAN
                                         (see signature 7.17)
```

The monitored function `interrupted` shown in signature 7.18 indicates if the given activity in a given instance has been interrupted.

```
monitored interrupted : ACTIVITIES × INSTANCES → BOOLEAN
                                         (see signature 7.18)
```

The controlled function `lifeCycleState` shown in signature 7.19 gives the instance life-cycle state of an activity [1, tab. 10.4]. Each time a token arrives on an incoming sequence flow of an activity a new instance of the target activity will be created with the initial life-cycle state “Ready”. The `lifeCycleState` function may

take one of the following values in case no errors or interruptions occur. During the run of one instance of an activity the life-cycle starts with the initial “Ready” state. Next, the activity will transit to the “Active” state when the data conditions are met. After the activity’s work was completed it will transit to the “Completing” state. Then, if the activity was not interrupted, it will transit to the “Completed” state after all completing requirements are met and assignments are completed. Finally, if no compensation occurred the activity will finish in the “Closed” state. In case of an error, interruption or compensation during the activity instance life-cycle run the activity may be found in one of the following life-cycle states: “Failing”, “Terminating”, “Compensating”, “Failed”, “Terminated”, “Compensated” and “Withdrawn”. For detailed information about the activity life-cycle states and the transitions between them see [1, sec. 13.2.2].

```
controlled lifeCycleState : ACTIVITIES × INSTANCES →
LIFE_CYCLE_STATES
```

(see signature 7.19)

The controlled function `instantiatingFlowNode` shown in signature 7.20 holds a flow node for which the given instance has been created. This is used, e.g., for sub-processes.

```
controlled instantiatingFlowNode : INSTANCES → FLOW-
_NODES
```

(see signature 7.20)

The abstract derived function `firingInstances : FLOW_NODES → MULTISET(INSTANCES)` shown in signature 7.7 defines a multiset of instances in which the given flow node fired. Since a flow node can fire in one step multiple times even in\* the same instance this function returns a multiset.

```
abstract derived firingInstances : FLOW_NODES →
MULTISET(INSTANCES)
```

(see signature 7.7)

The controlled function `parentInstance` shown in signature 7.21 holds the parent instance of the given instance. This may be `undef` for top-level process instances.

```
controlled parentInstance : INSTANCES → INSTANCES
```

(see signature 7.21)

The rule `InstanceTransition` : `ACTIVITIES` shown in listing 7.26 handles the life-cycle states of all relevant instances of the given activity using the `InstanceOperation` shown in signature 7.22.

```
rule InstanceTransition : ACTIVITIES [B.27]
```

```
1 rule InstanceTransition(activity) =
2   ∀ instance ∈ relevantInstances(activity) do
3     InstanceOperation(instance, activity)
```

The abstract rule `InstanceOperation` shown in signature 7.22 is responsible for instance related work. This rule is refined only for activities since other flow nodes cannot be instantiated.

```
abstract rule InstanceOperation : INSTANCES × ACTIVITIES
    (see signature 7.22)
```

The rule `ActivityTransition` shown in listing 7.27 refines the `control-Operation` for activities if the start quantity for enabling tokens is satisfied. Then first, an instance of the activity is created by consuming the enabling tokens. Afterwards, the instance of the activity is executed by the rule `GetActive` that calls the rule `StartOperation` which differentiates various types of activities. See [44] for detailed information. This will not be further refined in this thesis.

```
rule ActivityTransition : ACTIVITIES [B.28]
```

```
1 rule ActivityTransition(flowNode) =
2   if flowNode ∈ ACTIVITIES then
3     let firingInstances ← ExclusiveMergeBehavior(flowNode) in
4       FlowNodeTransition(flowNode) where
5         activationCondition(flowNode) = firingInstances ≠ 0
6         FlowNodeOperation(flowNode) =
7           ∀ instance ∈ firingInstances do
8             CreateInstance(flowNode, instance)
9
10      InstanceTransition(flowNode) where
11        InstanceOperation(instance, flowNode) = parblock
12        if lifeCycleState(instance, flowNode) = "Ready" then
13          GetActive(instance, flowNode)
14
15        if lifeCycleState(instance, flowNode) ∈ {"Completed",
16          "Compensated", "Failed", "Terminated", "Withdrawn"}
17        then parblock
18          ExitActivity(instance, flowNode)
19        endparblock
20      endparblock
```

The rule `CreateInstance` shown in listing 7.28 creates a new instance of an activity. Additionally, the new instance is added to the function `activeInstances : ACTIVITIES`. If a new instance of a top-level process is needed, then the `Dispatch` rule should be used. No token is created within this rule, because it will be created when the flow node is left.

```
rule CreateInstance : ACTIVITIES × INSTANCES [B.29]
```

```
1 rule CreateInstance(activity, parent) =
2   return instance in
3     instance ← new INSTANCES
4     add instance to activeInstances(activity)
5     instantiatingFlowNode(instance) ← activity
6     parentInstance(instance) ← parent
7     lifeCycleState(activity, instance) ← "Ready"
```

## B.6 Events

The static function `triggers` shown in signature 7.23 replaces the related attributes `eventDefinitions` and `eventDefinitionRefs` from the original meta-model [1, tab. 10.82]. From the execution point of view this function is the union of the two replaced attributes.

```
static triggers : EVENTS → SET(TRIGGERS)
                                     (see signature 7.23)
```

The static function `parallelMultiple` shown in signature 7.24 determines whether all defined triggers in an event have to occur in order the event fires or just one. It is only relevant when the catch event has more than one triggers defined [1, tab. 10.].

```
static parallelMultiple : CATCH_EVENTS → BOOLEAN
                                     (see signature 7.24)
```

The static function `attachedTo` shown in signature 7.25 denotes the activity that the given boundary event is attached to [1, tab. 10.91].

```
static attachedTo : BOUNDARY_EVENTS → ACTIVITIES
                                     (see signature 7.25)
```

The static function `conditionExpression` shown in signature 7.26 holds the condition expression for conditional triggers and timer triggers.

```
static conditionExpression : CONDITIONAL_TRIGGERS →
EXPRESSIONS
(see signature 7.26)
```

The controlled function `eventOccured` shown in signature 7.27 holds the information about event occurrence for a given flow node in a concrete instance.

```
controlled eventOccured : EVENTS × INSTANCES → BOOLEAN
(see signature 7.27)
```

The derived function `triggerName : EVENTS → STRINGS` shown in listing 7.29 checks `triggers` (see signature 7.23) of the given event and returns "None" if no trigger is defined for such an event. If more triggers are defined, it returns either "Multiple" or "ParallelMultiple" in case that the `parallelMultiple` (see signature 7.24) is set to true. Otherwise the appropriate trigger name ("Message", "Timer", "Error", etc.) is returned [1].

```
derived triggerName : EVENTS → STRINGS [B.30]
```

```
1 derived triggerName(event) =
2   return res in
3     if triggers(event) = 0 then
4       res ← "None"
5     else if |triggers(event)| = 1 then
6       choose trigger ∈ triggers(event) do
7         res ← triggerName(trigger)
8
9     else if parallelMultiple(event) then
10      res ← "ParallelMultiple"
11    else
12      res ← "Multiple"
```

The derived function `triggerName : TRIGGERS → STRINGS` shown in listing 7.30 returns the appropriate name ("Message", "Timer", "Error", etc.) for the given trigger [1].

```
derived triggerName : TRIGGERS → STRINGS [B.31]
```

```

1 derived triggerName( trigger) =
2   return res in
3     if trigger ∈ MESSAGE_TRIGGERS then
4       res ← "Message"
5     else if trigger ∈ TIMER_TRIGGERS then
6       res ← "Timer"
7     else if trigger ∈ ERROR_TRIGGERS then
8       res ← "Error"
9     else if trigger ∈ ESCALATION_TRIGGERS then
10      res ← "Escalation"
11    else if trigger ∈ CANCEL_TRIGGERS then
12      res ← "Cancel"
13    else if trigger ∈ COMPENSATION_TRIGGERS then
14      res ← "Compensation"
15    else if trigger ∈ CONDITIONAL_TRIGGERS then
16      res ← "Conditional"
17    else if trigger ∈ LINK_TRIGGERS then
18      res ← "Link"
19    else if trigger ∈ SIGNAL_TRIGGERS then
20      res ← "Signal"
21    else if trigger ∈ TERMINATE_TRIGGERS then
22      res ← "Terminate"

```

The EventTransition : EVENTS rule shown in listing 7.31 refines the WorkflowTransition : FLOW\_NODES rule shown in listing 7.4 and defines the activationCondition using the SelectFiringInstances rule shown in listing 7.32 and the FlowNodeOperation using an abstract ControlOperation for operations, i.e., production of tokens on outgoing sequence flows and an abstract EventOperation, i.e., throwing an event.

<pre>rule EventTransition : EVENTS</pre>	[B.32]
--	--------

```

1 rule EventTransition(event) =
2   let firingInstances ← SelectFiringInstances(event) in
3     WorkflowTransition(event) where
4       activationCondition(event) = firingInstances ≠ ∅
5     FlowNodeOperation(event) =
6       ControlOperation(event, firingInstances)
7     EventOperation(event, firingInstances)

```

The rule SelectFiringInstances shown in listing 7.32 separates the control condition from event condition for events. This separation is defined from here since events are the only flow nodes which actually give the event condition use and therefore it is not necessary to model this separation before.

The ControlActivationBehavior is responsible for the control condition and consuming tokens. Since tokens, which satisfy the control condition are consumed immediately the instances in which a token configuration satisfied the control condition will be cached using the waitingControlInstances function. Similarly occurred events will be immediately consumed and the instances in which they occurred will be cached using the waitingEventInstances function.

There are three possibilities of what can happen. First, the concrete event type waits for both the control and the event condition to be satisfied, i.e., the event type defines both the `ControlActivationBehavior` and the `EventActivationBehavior`. In this case the selected instances, which can actually fire the given event, are a union of the two cached instance sets. In other words the event can fire only in instances in which both, the control condition and the event condition, were satisfied.

The second and third possibility is that either only the control or the event condition needs to be satisfied. I.e., only one of the two behaviors (`ControlActivationBehavior` or `EventActivationBehavior`) is defined and the other yields `undef`. In this case also the related cache function yields `undef` and is ignored and only instances in the other cache function are relevant.

```
rule SelectFiringInstances : EVENTS → MULTISSET(-
INSTANCES) [B.33]
```

```

1 rule SelectFiringInstances(event) =
2   local allWaitingControlInstances ← waitingControlInstances(event)
3     ∪ ControlActivationBehavior(event),
4     allWaitingEventInstances ← waitingEventInstances(event)
5     ∪ EventActivationBehavior(event) in
6   return res in
7     if waitingControlInstances(event) ≠ undef
8       ∧ waitingEventInstances(event) ≠ undef then
9       res ← allWaitingControlInstances
10        ∩ allWaitingEventInstances
11     else if waitingControlInstances(event) ≠ undef
12       ∧ waitingEventInstances(event) = undef then
13       res ← allWaitingControlInstances
14     else if waitingControlInstances(event) = undef
15       ∧ waitingEventInstances(event) ≠ undef then
16       res ← allWaitingEventInstances
17     else
18       res ← ∅
19
20   waitingControlInstances(event) ← allWaitingControlInstances \ res
21   waitingEventInstances(event) ← allWaitingEventInstances \ res

```

The rule `CatchEventTransition : CATCH_EVENTS` shown in listing 7.33 refines the rule `EventTransition : EVENTS` (see listing 7.31) and defines only the event related parts of a flow node. A catch event defines `EventActivationBehavior` by checking if the event defined by a trigger related to the given flow node occurred. This is realized with the `eventOccurred : FLOW_NODES × INSTANCES → BOOLEAN` function (see signature 7.27). A catch event does not perform any `EventOperation` since no event is further thrown in this type of event node.

<code>rule CatchEventTransition : CATCH_EVENTS</code>	[B.34]
---	--------

```

1 rule CatchEventTransition(event) =
2   EventTransition(event) where
3     EventActivationBehavior(event) =
4       return res in
5         let instances ← { i | i ∈ INSTANCES ∧ eventOccured(event, i) } in
6           ∀ instance ∈ instances do
7             eventOccured(event, instance) ← ⊥
8           res ← instances
9
10  EventOperation(event, instances) = skip

```

The rule `ThrowEventTransition : THROW_EVENTS` shown in listing 7.34 refines the rule `EventTransition : EVENTS` (see listing 7.31) and defines only the event related parts of a flow node. A throw event is never activated by an event and therefore the `EventActivationBehavior` yields an `undef`. On the other hand, a throw event does perform `EventOperation` using `Throw` rule (see signature 7.28) in all instances it was fired.

<code>rule ThrowEventTransition : THROW_EVENTS</code>	[B.35]
---	--------

```

1 rule ThrowEventTransition(event) =
2   EventTransition(event) where
3     EventActivationBehavior(event) = undef
4     EventOperation(event, instances) =
5       ∀ instance ∈ instances do
6         ∀ trigger ∈ triggers(event) do
7           Throw(trigger, instance)

```

The abstract rule `Throw : TRIGGERS × INSTANCES` shown in signature 7.28 is meant to be fired for every trigger of throw event in a given instance.

<code>abstract rule Throw : TRIGGERS × INSTANCES</code>	(see signature 7.28)
---	----------------------

The rule `Throw : EXCEPTION_TRIGGERS × INSTANCES` shown in signature B.36 implements throwing of an error or an escalation trigger.

<code>rule Throw : EXCEPTION_TRIGGERS × INSTANCES</code>	[B.36]
--	--------

```

1 rule Throw(trigger, inst) =
2   let notification in

```



```

3  if trigger ∈ ERROR_TRIGGERS then
4    notification ← new ERROR_NOTIFICATIONS
5  else
6    notification ← new ESCALATION_NOTIFICATIONS
7
8  choose ctx ∈ CONTEXTS with instance(ctx) = inst do
9    context(notification) ← ctx
10   code(notification) ← code(trigger)
11   name(notification) ← name(trigger)

```

The `StartEventTransition : START_EVENTS` show in listing 7.35 refines the rule `CatchEventTransition : CATCH_EVENTS` (see listing 7.33), which defines the event related condition and operation. Whether the given flow node (event in this case) fires depends in this case only on the event condition and therefore the `ControlActivationBehavior` yields `undef`.

If a start event is a source of multiple outgoing sequence flows [1, sec. 10.4.2] and such sequence flows then form parallel paths, we reuse the `ParallelSplitBehavior : FLOW_NODES × INSTANCE` (see listing 7.20) to realize this. Additionally a start event creates a new instance of the encompassing activity.

<b>rule</b> StartEventTransition : START_EVENTS	[B.37]
---	--------

```

1 rule StartEventTransition(event) =
2   CatchEventTransition(event) where
3     ControlActivationBehavior(event) = undef
4     ControlOperation(event, instances) =
5       ∀ parent ∈ instances do
6         let instance ← CreateInstance(parentFlowNode(event), parent) in
7           ParallelSplitBehavior(event, instance)

```

The rule `EndEventTransition : END_EVENTS` shown in listing 7.36 refines the rule `ThrowEventTransition : THROW_EVENTS` (see listing 7.34), which defines the event related condition and operation. Whether the given event fires depends in this case also on the control condition which is defined as `ExclusiveMergeBehavior` (see listing 7.11). A end event has no outgoing sequence flows and therefore the `ControlOperation` does not perform anything.

<b>rule</b> EndEventTransition : END_EVENTS	[B.38]
---	--------

```

1 rule EndEventTransition(event) =
2   ThrowEventTransition(event) where
3     ControlActivationBehavior(event) = ExclusiveMergeBehavior(event)
4     ControlOperation(event, instances) = skip

```

The rule `IntermediateEventTransition : INTERMEDIATE_EVENTS` shown in listing 7.37 refines the rule `EventTransition : EVENTS` (see listing 7.31) and defines the control condition by defining the `ControlActivation-`

Behavior as ExclusiveMergeBehavior (see listing 7.11 and the Control-Operation using the ParallelSplitBehavior (see listing 7.20 for every firing instance).

The event related condition and operation will be further defined in subrules IntermediateCatchEventTransition (see listing 7.38) and IntermediateThrowEventTransition (see listing 7.39).

```
rule IntermediateEventTransition : INTERMEDIATE-
    _EVENTS [B.39]
```

```
1 rule IntermediateEventTransition(event) =
2   EventTransition(event) where
3     ControlActivationBehavior(event) = ExclusiveMergeBehavior(event)
4     ControlOperation(event, instances) =
5       ∀ instance ∈ instances do
6         ParallelSplitBehavior(event, instance)
```

The rule IntermediateCatchEventTransition : INTERMEDIATE-CATCH\_EVENTS shown in listing 7.38 is defined by the rule CatchEventTransition : CATCH\_EVENTS (see listing 7.33), which specifies the event condition and operation, and the rule IntermediateEventTransition : -INTERMEDIATE\_EVENTS (see listing 7.37), which specifies the control condition and operation.

```
rule IntermediateCatchEventTransition : INTERMEDIATE-
    _CATCH_EVENTS [B.40]
```

```
1 rule IntermediateCatchEventTransition(event) =
2   CatchEventTransition(event)
3   IntermediateEventTransition(event)
```

The rule IntermediateThrowEventTransition : INTERMEDIATE-THROW\_EVENTS shown in listing 7.39 is defined by the rule ThrowEventTransition : THROW\_EVENTS (see listing 7.34), which specifies the event condition and operation, and the rule IntermediateEventTransition : -INTERMEDIATE\_EVENTS (see listing 7.37), which specifies the control condition and operation. the control condition and operation.

```
rule IntermediateThrowEventTransition : INTERMEDIATE-
    _THROW_EVENTS [B.41]
```

```
1 rule IntermediateThrowEventTransition(event) =
2   ThrowEventTransition(event)
3   IntermediateEventTransition(event)
```

The rule `BoundaryEventTransition : BOUNDARY_EVENTS` shown in listing 7.40 refines the rule `CatchEventTransition : CATCH_EVENTS` (see listing 7.33) and defines the missing control condition and operation. The boundary event is not a target of any incoming sequence flows and therefore it does not wait for any tokens. This is modeled by the `ControlActivationBehavior` yielding `undef`. The boundary event can be source of one or more sequence flows in which case it will produce tokens on each of them in case the event fires. This is modeled using the `ParallelSplitBehavior` (see listing 7.20).

```
rule BoundaryEventTransition : BOUNDARY_EVENTS [B.42]
```

```
1 rule BoundaryEventTransition(event) =
2   CatchEventTransition(event) where
3     ControlActivationBehavior(event) = undef
4     ControlOperation(event, instances) =
5       ∀ instance ∈ instances do
6         ParallelSplitBehavior(event, instance)
```

## B.7 Gateways

The static function `gateConditionExpression` shown in signature 7.29 represents the gating condition for sequence flows. A token will only pass the gate on its outgoing sequence flow if this condition holds. In the original BPMN 2.0 specification this was implemented in sequence flows as `conditionExpression → EXPRESSIONS` [68, tab. 8.51], but since the gating refinement this parameter was renamed for better indication of its meaning and the placement of the actual condition.

```
static gateConditionExpression : SEQUENCE_FLOWS →
EXPRESSIONS
(seesignature7.29)
```

The static function `activationConditionExpression` shown in signature 7.30 determines which combination of incoming tokens will be synchronized for activation of the gateway [1, tab. 10.125].

```
static activationConditionExpression : COMPLEX_GATEWAYS
→ EXPRESSIONS
(seesignature7.30)
```

The internal state of a complex gateway is represented by the controlled function `waitingForStart` shown in signature 7.31 [1, tab. 10.126].

```
controlled waitingForStart : COMPLEX_GATEWAYS × INSTANCES
→ BOOLEAN
```

(see signature 7.31)

The controlled function `sequenceFlowsToIgnoreDuringReset` shown in signature 7.32 holds a set of sequence flows, which activated the complex gateway and should be ignored during the reset inclusive behavior.

```
controlled sequenceFlowsToIgnoreDuringReset : COMPLEX-
_GATEWAYS × INSTANCES → SET(SEQUENCE_FLOWS)
```

(see signature 7.32)

The rule `GatewayTransition` shown in signature B.43 represents the common refinement of `WorkflowTransition` (see listing 7.4) for the gateway flow node.

```
rule GatewayTransition : GATEWAYS
```

[B.43]

```
1 rule GatewayTransition(gateway) =
2   FlowNodeTransition(gateway)
```

The rule `DataBasedGatewayTransition : DATA_BASED_GATEWAYS` shown in listing 7.42 refines the rule `FlowNodeTransition : FLOW_NODES` (see listing 7.25) and defines the `controlCondition : FLOW_NODES`, `eventCondition : FLOW_NODES`, `ControlOperation : FLOW_NODES`, and `EventOperation : FLOW_NODES`. The event related refinements are not relevant for a gateway and therefore the derived function `eventCondition` always returns `true`, while the rule `EventOperation` does not perform anything.

The derived function `controlCondition : FLOW_NODES` defines the condition the gateway needs to hold in order to fire. This is realized using the `MergeBehavior : FLOW_NODES` (see signature 7.8). The concrete gateway type will then use one of the defined `ParallelMergeBehavior : FLOW_NODES`, `ExclusiveMergeBehavior : FLOW_NODES`, `InclusiveMergeBehavior : FLOW_NODES`, or `ComplexMergeBehavior : FLOW_NODES`.

The output part is realized by refining the rule `ControlOperation : FLOW_NODES`. There one of the relevant tokens, which contributed to firing the gateway, will be chosen to determine the instance of the running process. Any of the tokens in the `enablingTokens` set may be chosen, since by design all of them have to be from the same instance. This instance will then be passed to the rule `SplitBehavior : FLOW_NODES × INSTANCES` to perform the token production on

the outgoing sequence flows of the gateway. The concrete gateway type will then use one of the defined `ExclusiveSplitBehavior : FLOW_NODES × INSTANCES` or `InclusiveSplitBehavior : FLOW_NODES × INSTANCES`.

```
rule DataBasedGatewayTransition : DATA_BASED-
  _GATEWAYS [B.44]
```

```
1 rule DataBasedGatewayTransition(gateway) =
2   let firingInstances ← GatewayBehavior(gateway) in
3   GatewayTransition(gateway) where
4     activationCondition(gateway) = firingInstances ≠ 0
5     FlowNodeOperation(gateway) = GatewayOperation(gateway, firingInstances)
```

The rule `ParallelGatewayTransition : PARALLEL_GATEWAYS` shown in listing 7.43 defines the transition for parallel gateways by refining the rule `DataBasedGatewayTransition : DATA_BASED_GATEWAYS` (see listing 7.42) and specifying the `MergeBehavior` to be the `ParallelMergeBehavior` and the `SplitBehavior` to be the `ParallelSplitBehavior`.

```
rule ParallelGatewayTransition : PARALLEL_GATEWAYS [B.45]
```

```
1 rule ParallelGatewayTransition(gateway) =
2   DataBasedGatewayTransition(gateway) where
3     GatewayBehavior(gateway) = InputBehavior(gateway) where
4       MergeBehavior(gateway) = ParallelMergeBehavior(gateway)
5
6     GatewayOperation(gateway, instances) =
7       OutputBehavior(gateway, instances) where
8         SplitBehavior(gateway, instance) =
9           ParallelSplitBehavior(gateway, instance)
```

The rule `ExclusiveGatewayTransition : EXCLUSIVE_GATEWAYS` shown in listing 7.44 defines the transition for exclusive gateways by refining the rule `DataBasedGatewayTransition : DATA_BASED_GATEWAYS` (see listing 7.42) and specifying the `MergeBehavior` to be the `ExclusiveMergeBehavior` and the `SplitBehavior` to be the `ExclusiveSplitBehavior`.

```
rule ExclusiveGatewayTransition : EXCLUSIVE_GATEWAYS [B.46]
```

```
1 rule ExclusiveGatewayTransition(gateway) =
2   DataBasedGatewayTransition(gateway) where
3     GatewayBehavior(gateway) = InputBehavior(gateway) where
4       MergeBehavior(gateway) = ExclusiveMergeBehavior(gateway)
5
6     GatewayOperation(gateway, instances) =
```

```

7      OutputBehavior(gateway, instances) where
8      SplitBehavior(gateway, instance) =
9      ExclusiveSplitBehavior(gateway, instance)

```

The rule `InclusiveGatewayTransition : INCLUSIVE_GATEWAYS` shown in listing 7.45 defines the transition for inclusive gateways by refining the rule `DataBasedGatewayTransition : DATA_BASED_GATEWAYS` (see listing 7.42) and specifying the `MergeBehavior` to be the `InclusiveMergeBehavior` and the `SplitBehavior` to be the `InclusiveSplitBehavior`.

```

rule InclusiveGatewayTransition : INCLUSIVE_GATEWAYS [B.47]

```

```

1 rule InclusiveGatewayTransition(gateway) =
2   DataBasedGatewayTransition(gateway) where
3     GatewayBehavior(gateway) = InputBehavior(gateway) where
4       MergeBehavior(gateway) = InclusiveMergeBehavior(gateway)
5
6   GatewayOperation(gateway, instances) =
7     OutputBehavior(gateway, instances) where
8       SplitBehavior(gateway, instance) =
9       InclusiveSplitBehavior(gateway, instance)

```

The rule `ComplexGatewayTransition : COMPLEX_GATEWAYS` shown in listing 7.46 defines the transition for complex gateways by refining the rule `DataBasedGatewayTransition : DATA_BASED_GATEWAYS` (see listing 7.42) and specifying the `MergeBehavior` to be the `ComplexMergeBehavior` and the `SplitBehavior` to be the `InclusiveSplitBehavior`. The one difference of the `SplitBehavior` of complex gateway and inclusive gateway is that the `gate-ConditionExpression` may additionally guard the current state of the complex gateway [1, tab. 13.5].

```

rule ComplexGatewayTransition : COMPLEX_GATEWAYS [B.48]

```

```

1 rule ComplexGatewayTransition(gateway) =
2   DataBasedGatewayTransition(gateway) where
3     GatewayBehavior(gateway) = InputBehavior(gateway) where
4       MergeBehavior(gateway) = ComplexMergeBehavior(gateway)
5
6   GatewayOperation(gateway, instances) =
7     OutputBehavior(gateway, instances) where
8       SplitBehavior(gateway, instance) =
9       InclusiveSplitBehavior(gateway, instance)

```

## B.8 Notifications

The function code : `EXCEPTION_NOTIFICATIONS → STRINGS` shown in signature B.49 holds the code of a error notification or escalation notification.

controlled code : EXCEPTION\_NOTIFICATIONS → STRINGS [B.49]

The function name : SIGNAL\_NOTIFICATIONS → STRINGS shown in signature B.50 holds the name of a signal notification or message notification used among others to match the notification with a concrete catch event.

controlled name : SIGNAL\_NOTIFICATIONS → STRINGS [B.50]

The function name : EXCEPTION\_NOTIFICATIONS → STRINGS shown in signature B.51 holds the name of a error notification or escalation notification.

controlled name : EXCEPTION\_NOTIFICATIONS → STRINGS [B.51]

The function context shown in signature 8.2 holds the context where the given notification happened.

shared context : NOTIFICATIONS → CONTEXTS  
(see signature 8.2)

The function flowNode shown in signature 8.3 holds a concrete flow node the notification is meant for. This may be undef in many cases and is meant, e.g., for starting a new process instance, where selecting a concrete start event is desirable.

shared flowNode : NOTIFICATIONS → FLOW\_NODES  
(see signature 8.3)

The function payload shown in signature B.52 contains the payload of a message.

shared payload : MESSAGE\_NOTIFICATIONS → STRINGS [B.52]

The function signalPool shown in signature B.53 holds a set of signals arrived from the environment.

`shared signalPool → SET(SIGNALS)`

[B.53]

The function `messagePool` shown in signature B.54 holds a set of messages arrived from the environment.

`shared messagePool → SET(MESSAGES)`

[B.54]

The function `notifications` shown in signature B.55 is a priority queue holding all the notifications.

`monitored notifications → PRIORITY_QUEUE`

[B.55]

The function `occurrenceTime` shown in signature 8.4 holds the time the given notification occurred.

`monitored occurrenceTime : NOTIFICATIONS → TIME`  
(see signature 8.4)

Signals arriving from the environment are converted to notifications by the rule `ProcessSignalPool`, shown in listing 8.5.

`rule ProcessSignalPool`

[B.56]

```
1 rule ProcessSignalPool =  
2   ∀ signal ∈ signalPool do  
3     let notification ← new SIGNAL_NOTIFICATIONS in  
4       context(notification) ← "StaticContext"  
5       name(notification) ← name(signal)  
6  
7   signalPool ← ∅
```

Messages arriving from the environment are converted to notifications by the rule `ProcessMessagePool`, shown in listing 8.4.

`rule ProcessMessagePool`

[B.57]



```

1 rule ProcessMessagePool =
2   ∀ message ∈ messagePool do
3     let notification ← new MESSAGE_NOTIFICATIONS in
4       context(notification) ← "StaticContext"
5       name(notification) ← name(message)
6       payload(notification) ← payload(message)
7
8   messagePool ← ∅

```

The rule PublishNotification shown in listing 8.6 implement the publication forwarding concept [1, sec. 10.4.1].

```
rule PublishNotification : SIGNAL_NOTIFICATIONS [B.58]
```

```

1 rule PublishNotification(notification) =
2   //ForwardNotification(notification) where
3   local c ← context(notification) in
4     ∀ event ∈ CATCH_EVENTS do
5       ∀ trigger ∈ triggers(event) ∩ SIGNAL_TRIGGERS with
6         name(trigger) = name(notification) do
7         if trigger ∉ MESSAGE_TRIGGERS then
8           let duplicate ← Clone(notification) in
9             flowNode(duplicate) ← event
10
11       else
12         flowNode(notification) ← event
13
14     ∀ task ∈ RECEIVE_TASKS do
15       flowNode(notification) ← task

```

Publish only yet unassigned notifications:

```

18 if flowNode(notification) = undef then
19   ∀ child ∈ CONTEXTS with parentContext(child) = c do
20     let duplicate ← Clone(notification) in
21       context(duplicate) ← child
22
23   remove notification from notifications // lifetime ends

```

The rule PropagateNotification shown in listing 8.8 implements the propagation resolution forwarding concept [1, sec. 10.4.1].

```
rule PropagateNotification : EXCEPTION_NOTIFICATIONS [B.59]
```

```

1 rule PropagateNotification(notification) =
2   //ForwardNotification(notification) where
3   choose boundary ∈ BOUNDARY_EVENTS with
4     attachedTo(boundary) =
5       instantiatingFlowNode(instance(context(notification)))
6   ∧ ( ∃ trigger ∈ (triggers(boundary) ∩ EXCEPTION_TRIGGERS) with
7     code(trigger) = code(notification) do

```

```

8     flowNode(notification) ← boundary
9
10    if flowNode(notification) = undef then
11        let parent ← parentContext(context(notification)) in
12            if parent ∈ SUB_CONTEXTS then
13                context(notification) ← parent
14            else if parent ∈ ROOT_CONTEXTS
15                ∧ notification ∈ ERROR_NOTIFICATIONS then
16                Terminate
17
18    // If no parent, we already terminated and are in static context

```

The rule `ThrowImplicitNotification` shown in listing 8.3, is responsible for observing conditional and timer trigger and throw corresponding notifications if their conditions were met. The assumption made here is that every timer trigger can be generalized to a conditional trigger and the attributes: `timeDate`, `timeCycle` and `timeDuration` [1, tab. 10.101] can be expressed by a condition [1, tab. 10.95].

rule <code>ThrowImplicitNotifications</code>	[B.60]
--	--------

```

1 rule ThrowImplicitNotifications =
2   ∀ event ∈ CATCH_EVENTS do
3     ∀ trigger ∈ triggers(event) with
4       trigger ∈ CONDITIONAL_TRIGGERS do
5       ∀ i ∈ INSTANCES with evaluate(conditionExpression(trigger), i) = T do
6         choose ctx ∈ CONTEXTS with instance(ctx) = i do
7           let notification ← new NOTIFICATIONS in
8             context(notification) ← ctx
9             flowNode(notification) ← event

```

## B.9 Contexts

The function `instance` shown in signature B.61 returns an instance of a given context.

<code>monitored instance : CONTEXTS → INSTANCES</code>	[B.61]
--	--------

The function `parentContext` shown in signature 8.1 returns a parent context of the given context. If the given context is the static context of the running WFE than this function returns `undef`.

<code>monitored parentContext : CONTEXTS → CONTEXTS</code> <div style="text-align: right;">(see signature 8.1)</div>
---

The derived function `subContexts` shown in listing 8.2 computes all sub contexts of the given context.

`derived subContexts : CONTEXTS → SET(CONTEXTS)` [B.62]

```

1 derived subContexts(parent) =
2   { context | context ∈ CONTEXTS ∧ parentContext(context) = parent }

```

`derived waitingTasks : CONTEXTS → SET(TASKS)` [B.63]

```

1 derived waitingTasks(context) =
2   return res in
3     let parentI ← instance(context) in
4       local subC ← subContexts(context),
5         waiting ← { t | t ∈ flowNodes(instantiatingFlowNode(parentI))
6           ∧ t ∈ TASKS
7           ∧ { i | i ∈ activeInstances(t)
8             ∧ parentInstance(i) = parentI
9             ∧ completed(t, i) = ⊥ } } in
10
11     while subC ≠ ∅ do
12       choose childContext ∈ subC do
13         waiting ← waiting ∪ waitingTasks(childContext)
14       remove childContext from subC
15
16     res ← waiting

```

## B.10 Workflow Interpreter

The monitored function `abortedByEnvironment` shown in signature B.64 indicates whether all running instances of all processes should be aborted.

`monitored abortedByEnvironment → BOOLEAN` [B.64]

The function `activeInstances : ACTIVITIES → SET(INSTANCES)` shown in signature B.65 holds a list of active instances of given activity.

`controlled activeInstances : ACTIVITIES → SET(-  
INSTANCES)` [B.65]

The function `activeInstances : PROCESSES → SET(INSTANCES)` shown in signature B.66 holds a list of active instances of given process.

<code>controlled activeInstances : PROCESSES → SET(- INSTANCES)</code>	[B.66]
--	--------

The controlled function `instantiatingProcess` shown in signature B.67 holds a process for that the given instance has been created, e.g., top-level process.

<code>controlled instantiatingProcess : PROCESSES → FLOW- _NODES</code>	[B.67]
---	--------

The monitored function `newDeploymentsRequestedByEnvironment` shown in signature B.68 holds all new deployments requested by the environment.

<code>monitored newDeploymentsRequestedByEnvironment → SET(- DEPLOYMENTS)</code>	[B.68]
--	--------

The controlled function `deployments` shown in signature B.69 holds all deployed deployments to the WFI.

<code>controlled deployments → SET(DEPLOYMENTS)</code>	[B.69]
--	--------

The monitored function `expressionLanguages` shown in signature B.70 holds all used expression languages in a deployment.

<code>monitored expressionLanguages : DEPLOYMENTS → SET(- EXPRESSION_LANGUAGES)</code>	[B.70]
--	--------

The rule `HandleNewDeployments` shown in listing 8.9 observes the monitored function `newDeploymentsRequestedByEnvironment` and adds newly loaded deployments to the deployment manager

<code>rule HandleNewDeployments</code>	[B.71]
--	--------

```

1 rule HandleNewDeployments =
2   ∀ deployment ∈ newDeploymentsRequestedByEnvironment do
3     ∀ expressionLanguage ∈ expressionLanguages(deployment) do
4       CheckExpressionLanguage(expressionLanguage)
5

```

```

6  ∀ trigger ∈ triggers(deployment) do
7    CheckTrigger(trigger)
8
9  if ¬ deployment ∈ deployments then
10    add deployment to deployments
11
12  remove deployment from newDeploymentsRequestedByEnvironment

```

The rule `WorkflowTransitionInterpreter` shown in signature B.72 represents a WFI for a given process [44].

<b>rule</b> <code>WorkflowTransitionInterpreter</code> : PROCESSES [B.72]
---

```

1 rule WorkflowTransitionInterpreter(process) =
2   if ¬ abortedByEnvironment then
3     parblock
4       activeInstances(process) ← activeInstances(process)
5       \ { i | i ∈ activeInstances(process) ∧ ¬ stillActive(i) }
6
7       if triggerOfNewInstanceRequestedByEnvironment(process) ≠ undef then
8         Dispatch(process, triggerOfNewInstanceRequestedByEnvironment(process))
9         triggerOfNewInstanceRequestedByEnvironment(process) ← undef
10
11      ∀ flowNode ∈ flowNodes(process) do
12        WorkflowTransition(flowNode)
13
14    endparblock

```

The rule `Dispatch` shown in listing 8.10 is a modified `CreateInstance` responsible for dispatching – creating an instance and throwing a given trigger – a top-level process.

<b>rule</b> <code>Dispatch</code> : PROCESSES × TRIGGERS [B.73]
---

```

1 rule Dispatch(process, trigger) =
2   let instance ← new INSTANCES in
3     add instance to activeInstances(process)
4     instantiatingProcess(instance) ← process
5     lifeCycleState(instance, process) ← "Ready"
6     Throw(trigger, instance)

```



## Appendix C

# ASM Ground Model $\LaTeX$ package

This chapter describes the ASM Ground Model  $\LaTeX$  package developed as a support tool for writing this document with embedded ASM code covering the ASM code management. The problem lies in the fact that signatures, descriptions and also implementation of ASM rules and functions are usually written directly into the document or copy-pasted from one document to another. Such an approach tends to result in inconsistent code, where changes become a nightmare as soon as such specifications grows over certain, relatively small, size and it is almost impossible to keep all relevant documents up to date with each other. The basic idea was to separate the ASM of the ground model from the rest of the informal text of a specification. This allows first the use of ASM tools developing and possibly simulating the code. Such an ASM code is directly embeddable into the informal document, and can be possibly reused in more documents and a lot of time and effort spent on the code maintenance, especially in case of changes in the ground model, can be saved. This package solves the issue of embedding existing ASM code into  $\LaTeX$  documents. The user will be able to see and manage changes in both the documents and the ground model separately. The signatures will be consistent and unified over the whole document. Description of a concrete function or rule can be reused in all the documents if necessary. Referring to a rule or a function is easy as calling the  $\LaTeX$  commands described in section C.2.

### C.1 ASM code

In the first section we will deal with the ASM code, its structure and format.

#### C.1.1 Directory structure

For better orientation a package based directory structure is proposed. This means that the ground model may be logically separated into different packages. Every

package is represented by its own directory in the ground model root directory. The default directory structure configuration is shown in Figure C.1.

---

**Figure C.1** Directory Structure

---

```
|-- [ground model root]
|   |-- [package A]
|   |   |-- *.casm
|   |-- [package B]
|   |   |-- *.casm
|   |   ...
|   |-- [package X]
|   |   |-- *.casm
```

---

This file structure makes it easy to maintain in a file version system like Subversion [104] or GIT [105]. It reduces the amount of conflicts since the code is separated into packages, and functions and rules into separate files so users may make changes to only one package or file. In case a directory is empty, e.g., no such function or rule exists, it may be left out. In case the ground model will for any reason not be separated into packages then all the code should go to some kind of default package, e.g. “main”. If sub-packages are needed, e.g, in case of a more complex ground model, the parameter defining the package in the  $\text{\texttt{\textbackslash\texttt{MEX}}}$  commands will need to contain the whole sub path of the package (see section C.2).

### C.1.2 File naming conventions

Every function or rule has to be stored in a separate file. The name of the file is the name of the function or rule without input or output parameters and has to be stored in given directory based on the directory structure described in section C.1.1. The file names are case-sensitive. In case a function or rule has multiple different input/output parameter configuration but same name, then all such configurations will be stored in one file. The first function or rule in a file is the default one and the order of the rest is arbitrary. The default file extension is `*.casm`, but it may be changed using the `\asmCodeExt{[extension]}` command (see listing C.3). The only restriction is that all ASM source code files should have the same file extension.

### C.1.3 ASM file format conventions

Every function or rule must be documented. Even if the function or rule description may be empty the empty documentation block has to be present in the ASM source code file. The documentation was inspired by JavaDoc [106]. It has to begin with “`/**`” token and end with “`*/`” token on separate lines. The token “`/**`” token is not allowed to be used anywhere else than for the function or rule documentation begin. In case of a comment block only “`/*`” token may be used.



The documentation may contain following parts:

**description** should describe the functionality of the function or rule. It should be written in a way that it can be read without the code being present, i.e., avoiding relative reference expressions, e.g., “... this function ...”, etc. In case a function or rule has to be referenced the signature or the implementation figure can be referenced using `\ref` command or similar (for auto generated labels see section C.1.3.1 and C.1.3.2).  $\LaTeX$  commands may be used in the *description*. However, all ASM Ground Model  $\LaTeX$  package command, except of `\asmFormat` would cause an unexpected behavior and must not be used inside the *description* block.

**private description** can be written after the *description* separated with a “ \* --” line. This *private description* is ignored by the ASM Ground Model  $\LaTeX$  package command and serves only the purpose of notes or todos in the ASM code file.

**business signatures** is an optional list of business level signatures, which can be used in the  $\LaTeX$  document. The first one is treated as the default one and the rest should have an arbitrary but fixed order, since they are referenced by it.

**author** defines the author of the function or rule.

The different parts can be separated with an empty line from each other for better readability. Empty line in a documentation is a line containing only a space and a star (“ \*”) and generates a new paragraph if used in the *description* block otherwise it is ignored. The optional parts (usage, parameters, return, author), if present, are ignored by the commands of the ASM Ground Model  $\LaTeX$  package. The ordering of the annotated parts is arbitrary. Although, the *description* part must always come first, followed by the *private description*, if present. An example is shown in Figure C.2.

### C.1.3.1 Signatures

There are two types of ASM signatures: the declaration signature and the definition signature. Every, basic function, derived function and rule has a declaration signature, while only non-abstract derived functions and rules have definition signature. The difference is that a declaration signature defines the type, name and universes for all input and output parameters and definition signature defines type, name and names of all input parameters and is followed by the implementation of the derived function or rule. The declaration signature must be defined after the documentation block. The definition signature, if present, must be defined after the declaration signature. The function or rule signatures are built in the following way:

1. The keyword `abstract` (at the beginning of the signature) belongs to a derived function or rule and such derived function or rule is abstract.
2. *The next step differs for functions and rules:*

---

**Figure C.2** Documentation Example

---

```
1 /
2   The derived function \asmFormat{someExampleFunction} shown in
3   \autoreff{sig:asm:monitored someExampleFunction : U_A X U_B -> U_R}
4   receives two parameters, first from the universe \asmFormat{U_A}
5   and the second from the universe \asmFormat{U_B} and returns
6   a value from the universe \asmFormat{U_R}.
7
8   —
9
10  Private description not exported by the ASMGCM package commands.
11
12  @uses controlled function someOtherFunction : U_X -> U_R
13
14  @param U_A The first parameter
15  @param U_B The second parameter
16  @return U_R The universe of the return value
17
18  @business someExampleFunctionOfUaAndUb
19  @author John Doe <john.doe@example.com>
20 /
21 monitored someExampleFunction : U_A : U_B → U_R
```

---

**Rule:** • In case the rule in question is a main rule the keyword `main` will follow.

- Otherwise, the keyword `rule` will follow.

**Function:** one of: `controlled`, `monitored`, `out`, `shared`, `static` or `derived` keywords will follow depending on the function type.

3. Followed by the name of the function or rule. Naming convention differ for functions and rules: both, function and rule names are written in camel case. But, rules begin with an upper case letter and functions begin with a lower case letter.
4. If and only if a function or rule has one or more input parameters:
  - (a) A colon follows the function or rule name surrounded by one space on both sides.
  - (b) A list of universes of the parameters follows, separated using the symbol: “X” (upper case “x”) surrounded by one space from both sides.
5. If and only if a function or rule have one or more output parameters:
  - (a) The string: “->” follows (“slash/minus” sign followed by “greater than” sign) surrounded by one space at both sides.
  - (b) The list of universes of the output parameters follows, separated using the symbol: “X” (upper case “x”) surrounded by one space at both sides as in the case of input parameters.

In case there are two or more functions or rules with the same name but different parameters they will all be defined in the same file. All different signatures need their own documentation block. The first function or rule in a file is treated as *default* one. The rest may have arbitrary unfixed ordering. Some examples of different declaration signatures can be seen in Figure C.3.

---

**Figure C.3** Signature Examples

---

```

1 main rule ExampleOfMainRule
2 rule ExampleOfRuleWithInputParameters: U_A X U_B X U_C
3 abstract rule ExampleOfAbstractRule : U_A X U_B → U_R
4 derived exampleOfDericedFunction : U_A → U_R
5 controlled exampleOfControlledFunction : U_A → U_R
6 shared exampleOfSharedFunctionWithMultipleOutputParameters : U_A → U_RA X
    U_RB X U_RC

```

---

A signature has to be always on one line, even if it violates any line length conventions.

### C.1.3.2 Implementation

Non-abstract derived functions and rules must contain an, even empty, implementation. An example can be seen in Figure C.4. Every implementation begins with a definition signature, which is built similarly as the declaration signature described in section C.1.3.1:

1. *The first step differs for functions and rules:*

- Rule:**
- In case the rule in question is a main rule the keyword `main` will follow.
  - Otherwise, the keyword `rule` will follow.

**Function:** one of: `controlled`, `monitored`, `out`, `shared`, `static` or `derived` keywords will follow depending on the function type.

2. Followed by the name of the function or rule. Naming convention differ for functions and rules: both, function and rule names are written in camel case. But, rules begin with an upper case letter and functions begin with a lower case letter.
3. If and only if a function or rule has one or more input parameters:
  - (a) An open parenthesis (“(”) follows the function or rule name without any surrounding spaces.
  - (b) A list of parameter names used further in the implementation separated using commas and may be optionally followed by a space follow.
  - (c) A close parenthesis (“)”) follows the function or rule name without any surrounding spaces.

4. Signature ends with a space and equal sign (“ =”). Implementation follows from the next line.

Also the **implementation signature has to be always on one line**, even if it violates any line length conventions. After the implementation signature the own implementation follows, where the defined parameter names may be used, and every line of the implementation has to be indented.

---

**Figure C.4** Implementation Examples

---

```
1 monitored someExampleFunction(paramA, paramB) =  
2 return res in  
3   // own implementation  
4   res ← someOperation(paramA, paramB)
```

---

### C.1.3.3 Indentation

Two spaces should be used for indentation. Tabs must not be used since they behave differently in different editors and the goal is to be able to reuse the code in different documents without the need of any editing. Four spaces should be used in case a long expression does not fit in one line (following the line length convention described in section C.1.3.4) and needs to be continued on the next line.

### C.1.3.4 Line length constraints

Avoid lines longer than 80 characters inside the *implementation* block, since they're not handled well by many tools. The actual line length may be even shortened if the compiled document requires it. The only exceptions are both declaration and definition signatures. These line length constraints do not apply for the *documentation* block.

## C.2 $\text{\LaTeX}$ documents

In this section the ASM Ground Model  $\text{\LaTeX}$  package and inclusion of ASM code into  $\text{\LaTeX}$  documents will be described.

### C.2.1 Configuring the ASM Ground Model $\text{\LaTeX}$ package

The ASM Ground Model  $\text{\LaTeX}$  package was created for the purpose of including the ASM signatures, code and description to any  $\text{\LaTeX}$  document. This package provides some basic parsing functionality of the ASM source code files and text formatting of ASM code, descriptions and signatures. The package can be easily included to a  $\text{\LaTeX}$  document by the `\usepackage` command as shown in listing C.1. The package is assuming by default that the ASM source code root is in a directory denoted by a relative path code to the  $\text{\LaTeX}$  document. To change that, the command `\asmCodePrefix`

---

**Listing C.1** Using the ASM Ground Model  $\text{\LaTeX}$  package in a  $\text{\LaTeX}$  document

---

```
1 \usepackage{asmgm}
```

---

---

**Listing C.2** Defining the path to the ASM ground model

---

```
1 \asmCodePrefix{relative/path/to/code/root}
```

---

can be used in the  $\text{\LaTeX}$  document as shown in listing C.2. Further, the package is also assuming that the extension of the files containing ASM functions or rules is “\*.casm”. This can be changed using the `\asmCodeExt` command in the  $\text{\LaTeX}$  document as shown in listing C.3. Declaration signatures, which are not used in-line, are

---

**Listing C.3** Configuration of file extension

---

```
1 \asmCodeExt{newFileExtension}
```

---

by default preceded with the “**Signature:**” string. This can be changed using the `\asmSignaturePrefix` command in the  $\text{\LaTeX}$  document as shown in listing C.4.

---

**Listing C.4** Changing the prefix for declaration signatures

---

```
1 \asmSignaturePrefix{Some prefix}
```

---

Even more than one ASM ground model project can be used in one document. All the configuration commands take immediate effect and therefore can be reconfigured at any place in the document.

### C.2.2 $\text{\LaTeX}$ commands provided by the ASM Ground Model $\text{\LaTeX}$ package

The ASM Ground Model  $\text{\LaTeX}$  package provides the `\asmFormating` and `\asmFormat` commands to unify the formatting of the document.

`\asmFormating` takes exactly one required parameter, which is expected to be a declaration signature and reformats it in the following way:

- Replaces “ -> ” (space, followed by slash/minus sign, followed by greater than sign, followed by space) with “ $\rightarrow$ ”.
- Replaces “ X ” (space, followed by “X”, followed by space) with “ $\times$ ”.
- Puts a *OK to hyphenate a word here* command (“\-”) before every underscore (“\_”) and between any case change from lower case to upper-case.

`\asmFormat` takes two parameters. The first one is required and is expected to be a declaration signature and the second one is optional and if it takes the value “abstract” it enforces abstract formatting. Internally this command uses also the `\asmFormating` command to preformat the signature first and then it uses different font face for abstract and non-abstract signatures. If the signature contains the keyword “abstract” or the second parameter takes the value “abstract” then the font face for abstract signatures will be used. Otherwise the font face for non-abstract signatures will be used.

### C.2.3 Including ASM code to $\text{\LaTeX}$ documents

In addition the package enables commands shown in listing C.5, which can be used in a  $\text{\LaTeX}$  document. Where `package`, `type` (“rule” or function type), `name` refers to

---

**Listing C.5**  $\text{\LaTeX}$  commands for ASM inclusion into  $\text{\LaTeX}$  documents

---

```

1 \asm{package}{type}{name}[parameters][options]
2 \asmDescription{package}{type}{name}[parameters]
3 \asmName{package}{type}{name}[parameters]
4 \asmBusiness{package}{type}{name}[parameters][number]
5 \asmSig{package}{type}{name}[parameters]
6 \asmSignature{package}{type}{name}[parameters]
7 \asmImplementation{package}{type}{name}[parameters][options]
8 \asmFloat{package}{type}{name}[parameters][options][search]
9 \asmFile{package}{type}{name}

```

---

the file structure described in section C.1.1. The optional parameter `parameters` is the part of the signature followed after the colon containing both input and output parameters with all the separators as in the ASM source code file. This becomes useful in case more than one function or rule is defined in the same source file. In case this parameter is not set in the  $\text{\LaTeX}$  command, the first *default* definition of a function or rule in the source file will be used.

As described in section C.1.3 every function or rule definition in an ASM source file can be divided into following parts:

- documentation
  - description
  - private description (*optional*)
  - usage (*optional*)
  - input parameters (*optional*)
  - output parameters (*optional*)
  - business signatures (*optional*)
  - author (*optional*)
- declaration signature

- definition signature followed by implementation (*required only for non-abstract derived function and rules*)

In the code examples the following references will be used: [documentation\_description], [signature] and [implementation] refers to the file part of concrete function or rule described in section C.1.3. For the purpose of the examples, a rule example is placed in code/main/rules (Figure C.5) and a function example was placed in code/main/functions/derived (Figure C.6).

---

**Figure C.5** A rule example

---

```

1 /
2  Quisque commodo varius elit eget pellentesque. In hac habitasse platea
3  dictumst. Cras eu euismod velit.
4
5  Quisque cursus sagittis fermentum. Nam vulputate,
6  tellus vitae fringilla fermentum, ipsum est feugiat neque, vel gravida sem
7  nunc vitae orci.
8
9  Nunc facilisis dui eu felis vulputate imperdiet. Duis luctus pulvinar
10 molestie.
11
12 @param U_A Universe A
13 @param U_B Universe B
14 @param U_C Universe C
15 /
16 rule RuleExample : U_A X U_B X U_C
17 rule RuleExample(paramA, paramB, paramC) =
18 // the implementation of the rule RuleExample : U_A X U_B X U_C
19 // here ...
20 if statement then
21 / Quisque cursus sagittis fermentum. /
22 DoSomething(paramA, paramB)
23 else if other_statement then
24 / Curabitur nunc diam, scelerisque et ultricies quis, gravida eu
25 Cras eu \emph{euismod} velit.
26
27 Cras eu euismod velit. In id ante vitae lorem rhoncus ultricies.
28 Sed in orci ac elit rutrum vulputate. /
29 DoSomethingElse(paramA, paramC)
30 else
31 /
32 Curabitur nunc diam, scelerisque et ultricies quis, gravida eu
33 eros. Cras posuere nunc vitae sem molestie vel posuere arcu egestas.
34 /
35 DoOtherwise(paramC)

```

---

---

**Figure C.6** A monitored function example

---

```
1 /
2 Lorem ipsum dolor sit \textbf{amet}, consectetur adipiscing elit. Proin
3 nisl diam, gravida non tincidunt vitae, hendrerit quis urna.
4
5 Morbi vel ultricies nunc. Sed mauris lectus, tincidunt nec dignissim in,
6 fermentum ac nibh. Vivamus facilisis sodales lectus eu luctus.
7
8 @param U_A Universe A
9 @param U_B Universe B
10
11 @return U_R1 Return universe 1
12 @return U_R2 Return universe 2
13
14 @business functionExampleOfUaAndUb
15 @business functionExampleWithUaOfUb
16
17 @author John Doe <john.doe@example.com>
18 /
19 derived functionExample : U_A X U_B → U_R1 X U_R2
20 derived functionExample(paramA, paramB) =
21   // the implementation of the function
22   // functionExample : U_A X U_B → U_R1 X U_R2
23   // here ...
24   return [paramA, paramB]
25
26 /
27 Duis metus dui, viverra eget condimentum quis, gravida et nibh. Morbi metus
28 erat, dapibus et ullamcorper nec, elementum eget dui.
29
30 @business functionExampleOfUa
31 @author John Doe <john.doe@example.com>
32 /
33 abstract derived functionExample : U_A → U_R
34
35 /
36 Nunc ac sapien ligula. Nam ultricies urna vitae sapien rutrum ac laoreet
37 nunc congue. Mauris posuere tempor orci ac aliquam. Cras lacinia lectus eu
38 tellus vestibulum scelerisque. Proin vel consequat erat. In hac habitasse
39 platea dictumst. Nunc nibh leo, malesuada.
40
41 @business functionExampleGivenUaWithUb
42 @author John Doe <john.doe@example.com>
43 /
44 derived functionExample : U_A X U_B → U_R
45 derived functionExample(paramA, paramB) =
46   // the implementation of the function
47   // functionExample : U_A X U_B → U_R
48   // here ...
49   return someValue
```

---



### C.2.3.1 Descriptions of functions and rules in $\text{\LaTeX}$ documents

The  $\text{\LaTeX}$  command `\asmDescription` will read the corresponding file with the function or rule, extract the description and include it into the  $\text{\LaTeX}$  document as one or more paragraphs. The description can contain  $\text{\LaTeX}$  commands but there are some things the writer has to pay more attention to. First is that any opening bracket “{” has to be closed “}” on the same line. Otherwise the parsing will fail. Even in case the  $\text{\LaTeX}$  command parameter is longer than defined as the line length limit in section C.1.3.4, this convention has to be violated. In case “{” and “}” brackets are used in the implementation part of a function or rule and there is a need of having them on different lines, they have to be replaced with “( [” and “] )” strings. The ASM Ground Model  $\text{\LaTeX}$  package will transform the “( [” into a “{” and the “] )” into a “}” in the output document. Secondly, commands shown in listing C.5 must not be used. Only the `\asmFormatting` and `\asmFormat` commands from the ASM Ground Model  $\text{\LaTeX}$  package may be used in the description of a function or rule or in comments.

The `\asmDescription{package}{type}{name}[parameters]` takes up to 4 parameters, where `package`, `type` (“rule” or function type), `name` refers to the file structure described in section C.1.1. The optional parameter `parameters` is the part of the signature followed after the colon containing both input and output parameters with all the separators as in the ASM source code file. This becomes useful in case more than one function or rule is defined in the same source file. In case this parameter is not set in the  $\text{\LaTeX}$  command, the first *default* definition of a function or rule in the source file will be used.

**Example 1:** shows the documentation of `RuleExample` rule. The command `\asmDescription{main}{rule}{RuleExample}` will produces part of the document shown in Figure C.7.

---

**Figure C.7** Description of `RuleExample`

---

Quisque commodo varius elit eget pellentesque. In hac habitasse platea dictumst. Cras eu euismod velit. Quisque cursus sagittis fermentum. Nam vulputate, tellus vitae fringilla fermentum, ipsum est feugiat neque, vel gravida sem nunc vitae orci. Nunc facilisis dui eu felis vulputate imperdiet. Duis luctus pulvinar molestie.

---

**Example 2:** shows the documentation of the *default* `functionExample` function. The command `\asmDescription{main}{derived}{functionExample}` will produces part of the document shown in Figure C.8

---

**Figure C.8** Description of the default `functionExample`

---

Lorem ipsum dolor sit **amet**, consectetur adipiscing elit. Proin nisl diam, gravida non tincidunt vitae, hendrerit quis urna. Morbi vel ultricies nunc. Sed mauris lectus, tincidunt nec dignissim in, fermentum ac nibh. Vivamus facilisis sodales lectus eu luctus.

---

**Example 3:** shows the documentation of the `functionExample : U_A × U_B → U_R` function. The command `\asmDescription{main}{derived}{functionExample}[U_A × U_B → U_R]` will produce part of the document shown in Figure C.9

---

**Figure C.9** Description of `functionExample : U_A × U_B → U_R`

---

Nunc ac sapien ligula. Nam ultricies urna vitae sapien rutrum ac laoreet nunc congue. Mauris posuere tempor orci ac aliquam. Cras lacinia lectus eu tellus vestibulum scelerisque. Proin vel consequat erat. In hac habitasse platea dictumst. Nunc nibh leo, malesuada.

---

### C.2.3.2 Signatures of functions and rules in $\text{\LaTeX}$ documents

The  $\text{\LaTeX}$  command `\asmSignature{package}{type}{name}[parameters]`, `\asmSig{package}{type}{name}[parameters]`, `\asmBusiness{package}{type}{name}[parameters][number]` and `\asmName{package}{type}{name}[parameters]` will read the corresponding source file with the function or rule, extract the signature and include it into the  $\text{\LaTeX}$  document. The difference between `\asmSignature` and `\asmSig` is that `\asmSignature` includes the complete signature in a box taking the whole page width while the `\asmSig` is for including the signature in-line in the surrounding text. Additionally the `\asmSignature` will prepend the prefix defined using the `\asmSignaturePrefix` command. Such signature box can be placed multiple times in a  $\text{\LaTeX}$  document. The first time it is placed for a concrete declaration signature it will be numbered with a distinct counter and a label will be created for it in order to enable the possibility to reference such signature. The generated label will have the following format: `sig:asm:[signature]`, where the `[signature]` is a full signature including all keywords and universes as it is on the whole line in the ASM source code.

The `\asmName{package}{type}{name}[parameters]` command is meant for in-line use as the `\asmSig` command, but it prints out only the name of the function or rule without any parameter universes. The `\asmBusiness{package}{type}{name}[parameters][number]` command searches the documentation block for `@business` annotated entries and prints them in-line to the surrounding text. Such *business signature* is additionally formatted using the `\asmFormat` command.

The above command takes the following parameters: `package`, `type` (“rule” or function type), `name` refers to the file structure described in section C.1.1. The optional parameter `parameters` is the part of the signature followed after the colon containing both input and output parameters with all the separators as in the ASM source code file. This becomes useful in case more than one function or rule is defined in the same source file. In case this parameter is not set in the  $\text{\LaTeX}$  command, the first *default* definition of a function or rule in the source file will be used. Additionally, the `\asmBusiness{package}{type}{name}[parameters][number]` command takes a fifth optional parameter identifying the position of the *business signature*. By default the first is taken if this parameter is left out.

**Example 1:** shows the signature of `RuleExample` rule. The command `\asmSignature{main}{rule}{RuleExample}` will produce part of the document shown as sig-

nature C.1.

```
rule RuleExample : U_A × U_B × U_C [C.1]
```

**Example 2:** shows the signature of the *default* `functionExample` function. The command `\asmSignature{main}{derived}{functionExample}` will produce part of the document shown as signature C.2.

```
derived functionExample : U_A × U_B → U_R1 × U_R2 [C.2]
```

**Example 3:** shows the signature of `functionExample : U_A × U_B → U_R` function. The command `\asmSignature{main}{derived}{functionExample}[U_A X U_B → U_R]` will produce part of the document shown as signature C.3.

```
derived functionExample : U_A × U_B → U_R [C.3]
```

**Example 4:** shows the inline signature of `functionExample` function. The  $\TeX$  code shown in listing C.6 will result into a part of the document shown in Figure C.10.

---

**Listing C.6** Code producing inline signature of `functionExample`

---

```
1 ... some text before ...
2 \asmSig{main}{derived}{functionExample} dolor, interdum
3 ... ^ some text after ...
```

---



---

**Figure C.10** Inline signature of `functionExample`

---

```
... some text before ... derived functionExample : U_A × U_B → U_R1
× U_R2 dolor, interdum ... and some text after ...
```

---

### C.2.3.3 Implementations of functions and rules in $\TeX$ documents

The  $\TeX$  command `\asmImplementation{package}{type}{name}[parameters][options]`, where `package`, `type` (“rule” or function type), `name` refers to the file structure described in section C.1.1. The optional parameter `parameters` is the part of the signature followed after the colon containing both input and output parameters with all the separators as in the ASM source code file. This becomes useful in case more than one function or rule is defined in the same source file. In case this

parameter is not set in the  $\text{\texttt{\LaTeX}}$  command, the first *default* definition of a function or rule in the source file will be used. The second optional parameter can take the value “implementation” (default value) or “interspersed”. If this parameter takes the default value “implementation” or is left out the implementation will be printed into one listing environment. If the forth optional parameter takes the value “interspersed” the comment blocks — those, which begin with “/\*” and end with “\*/” — will be taken out and printed as a normal text. The  $\text{\texttt{\LaTeX}}$  command  $\text{\texttt{\backslash asmImplementation}}\{\text{package}\}\{\text{type}\}\{\text{name}\}[\text{parameters}][\text{options}]$  provided by the ASM Ground Model  $\text{\texttt{\LaTeX}}$  package, will read the corresponding file with the function or rule, extract the implementation and include it in the current  $\text{\texttt{\LaTeX}}$  document using  $\text{\texttt{\LaTeX}}$  package. Attention has to be paid to the “{”, “}” bracket problem described in section C.2.3.1.

For the purpose of implementation the ASM Ground Model  $\text{\texttt{\LaTeX}}$  package defines an ASM language for the listings package. This language defines following keywords: abstract, add, and, choose, clone, constraint, controlled, derived, diff, do, else, elsif, endif, enum, endpar, endparblock, endseq, endseqblock, false, forall, forsome, from, function, holds, if, in, is, let, local, main, monitored, new, or, out, par, parblock, remove, return, rule, select, seq, seqblock, shared, skip, static, then, to, true, undef, union, universe, where, while and with. Additionally, due to the fact that the ASM source code is a plain text and to improve readers experience, the ASM Ground Model  $\text{\texttt{\LaTeX}}$  package introduces translations to mathematical symbols shown in Table C.1 and wraps following symbols into a math environment: “{”, “}”, “(”, “)”, “[”, “]”, “>”, “<”, “+”, “-”, “|”, “=”, “,”, “;”, “?” and “/”.

Additionally the package provides  $\text{\texttt{\backslash asmFloat}}\{\text{package}\}\{\text{type}\}\{\text{name}\}[\text{parameters}][\text{options}]$  command taking the same set of parameters as the  $\text{\texttt{\backslash asmImplementation}}\{\text{package}\}\{\text{type}\}\{\text{name}\}[\text{parameters}][\text{options}]$ , does the same extraction and additionally wraps the implementation into a float environment, where the caption is a formatted signature of the function or rule and the float will be referable using auto-generated label in format: “fig:asm:[signature]”, where the [signature] is a full signature including all keywords and universes.

**Example 1:** shows the implementation of RuleExample rule. The command  $\text{\texttt{\backslash asmImplementation}}\{\text{main}\}\{\text{rule}\}\{\text{RuleExample}\}$  will produce the part of the document shown in Figure C.11.

**Example 2:** shows the implementation of RuleExample rule with interspersed comments. The command  $\text{\texttt{\backslash asmImplementation}}\{\text{main}\}\{\text{rule}\}\{\text{RuleExample}\}[\text{interspersed}]$  will do similar job like the one before but it will take out comments, which begin with “/\*” and end with “\*/” and produce the part of the document shown in Figure C.12.

**Example 3:** shows the implementation of the *default* functionExample function. The command  $\text{\texttt{\backslash asmImplementation}}\{\text{main}\}\{\text{derived}\}\{\text{functionExample}\}$  will produce the part of the document shown in Figure C.13.

**Table C.1** Translation table for ASM code

<code>+-</code>	$\pm$
<code>:</code>	$:$
<code>([</code>	$\{$
<code>])</code>	$\}$
<code>{}</code>	$\emptyset$
<code>:=</code>	$\leftarrow$
<code>!=</code>	$\neq$
<code>&lt;=</code>	$\leq$
<code>&gt;=</code>	$\geq$
<code>&lt;&lt;</code>	$\ll$
<code>&gt;&gt;</code>	$\gg$
<code>-&gt;</code>	$\rightarrow$
<code>&lt;-</code>	$\leftarrow$
<code>&lt;-&gt;</code>	$\leftrightarrow$
<code>...</code>	$\cdots$
<code>*</code>	$\cdot$
<code>infinity</code>	$\infty$

(a) basic symbols

<code>does_not_exist</code>	$\nexists$
<code>contains</code>	$\ni$
<code>diff</code>	$\setminus$
<code>forall</code>	$\forall$
<code>fornone</code>	$\nexists$
<code>forone</code>	$\exists!$
<code>forsome</code>	$\exists$
<code>intersect</code>	$\cap$
<code>is_in</code>	$\in$
<code>is_not_in</code>	$\notin$
<code>subset</code>	$\subset$
<code>subsesteq</code>	$\subseteq$
<code>sum</code>	$\sum$
<code>supset</code>	$\supset$
<code>supseteq</code>	$\supseteq$
<code>U</code>	$\cup$
<code>union</code>	$\cup$

(b) set operators

<code>!</code>	$\neg$
<code>and</code>	$\wedge$
<code>iff</code>	$\Leftrightarrow$
<code>not</code>	$\neg$
<code>or</code>	$\vee$
<code>par</code>	$\parallel$
<code>xor</code>	$\vee$
<code>true</code>	$\perp$
<code>false</code>	$\top$

(c) logic operators

<code>COMPLEX</code>	$\mathbb{C}$
<code>COMPLEX_NUMBERS</code>	$\mathbb{C}$
<code>INTEGERS</code>	$\mathbb{Z}$
<code>NATURALS</code>	$\mathbb{N}$
<code>NATURAL_NUMBERS</code>	$\mathbb{N}$
<code>RATIONALS</code>	$\mathbb{Q}$
<code>RATIONAL_NUMBERS</code>	$\mathbb{Q}$
<code>REALS</code>	$\mathbb{R}$
<code>REAL_NUMBERS</code>	$\mathbb{R}$

(d) universes

---

**Figure C.11** Implementation of RuleExample

---

```
1 rule RuleExample(paramA, paramB, paramC) =  
2 // the implementation of the rule RuleExample : U_A X U_B X U_C  
3 // here ...  
4 if statement then  
5 / Quisque cursus sagittis fermentum. /  
6 DoSomething(paramA, paramB)  
7 else if other_statement then  
8 / Curabitur nunc diam, scelerisque et ultricies quis, gravida eu  
9 Cras eu \emph{euismod} velit.  
10  
11 Cras eu euismod velit. In id ante vitae lorem rhoncus ultricies.  
12 Sed in orci ac elit rutrum vulputate. /  
13 DoSomethingElse(paramA, paramC)  
14 else  
15 /  
16 Curabitur nunc diam, scelerisque et ultricies quis, gravida eu  
17 eros. Cras posuere nunc vitae sem molestie vel posuere arcu egestas.  
18 /  
19 DoOtherwise(paramC)
```

---

---

**Figure C.12** Implementation of RuleExample with interspersed comments

---

```
1 rule RuleExample(paramA, paramB, paramC) =  
2 // the implementation of the rule RuleExample : U_A X U_B X U_C  
3 // here ...  
4 if statement then  
  
Quisque cursus sagittis fermentum.  
  
6 DoSomething(paramA, paramB)  
7 else if other_statement then  
  
Curabitur nunc diam, scelerisque et ultricies quis, gravida eu Cras eu euismod velit.  
Cras eu euismod velit. In id ante vitae lorem rhoncus ultricies. Sed in orci ac elit  
rutrum vulputate.  
  
13 DoSomethingElse(paramA, paramC)  
14 else  
  
Curabitur nunc diam, scelerisque et ultricies quis, gravida eu eros. Cras posuere  
nunc vitae sem molestie vel posuere arcu egestas.  
  
19 DoOtherwise(paramC)
```

---

---

**Figure C.13** Implementation of the *default* functionExample

---

```
1 derived functionExample(paramA, paramB) =  
2 // the implementation of the function  
3 // functionExample : U_A X U_B -> U_R1 X U_R2  
4 // here ...  
5 return [paramA, paramB]
```

---

**Example 4:** shows the implementation of `functionExample : U_A × U_B → U_R` function. The command `\asmImplementation{main}{derived}{functionExample}[U_A X U_B → U_R]` will produce the part of the document shown in Figure C.14.

---

**Figure C.14** Implementation of `functionExample : U_A × U_B → U_R`

---

```

1 derived functionExample(paramA, paramB) =
2   // the implementation of the function
3   // functionExample : U_A X U_B -> U_R
4   // here ...
5   return someValue

```

---

#### C.2.3.4 The whole function or rule in $\text{\LaTeX}$ documents

The ASM Ground Model  $\text{\LaTeX}$  package provides also a command allowing full inclusion of a function or rule into a  $\text{\LaTeX}$  document. This can be done using the `\asm{package}{type}{name}[parameters][options]` commands, where the parameters take same values as in case of `\asmImplementation{package}{type}{name}[parameters][options]` described in section C.2.3.3. This command prints the description of the function or rule followed by its signature and if such a function or rule has an implementation this will be printed out too. The fifth optional parameter takes additionally a “float” value, which makes the implementation be wrapped into a float environment. In case the fifth parameter needs to take more then one value, those should be separated by a comma. Some usage examples of the `\asm` command can be seen in Figure C.15.

---

**Figure C.15** Usage example of `\asm` command

---

```

1 \asm{main}{rule}{RuleExample}
2 \asm{main}{rule}{RuleExample}[] [interspersed]
3 \asm{main}{rule}{RuleExample}[] [float, interspersed]
4 \asm{main}{derived}{functionExample}
5 \asm{main}{derived}{functionExample}[] [interspersed]
6 \asm{main}{derived}{functionExample}[processes → Set]
7 \asm{main}{derived}{functionExample}[processes → Set] [float, interspersed]

```

---





## Appendix D

# Curriculum vitae



### About

Jan Kubový studied computer science at the Czech Technical University in Prague (CTU), where he also worked as a researcher in the Agent Technology Center (ATG). In parallel he studied software engineering and management at the Johannes Kepler University in Linz (JKU) as a double-degree program. Since 2011 he is a Ph.D. student at the Institute of Application Oriented Knowledge Processing (FAW). As a student of software engineering and mathematics he is now working on business processes and their formal description. In his master studies he was focusing on system and network security, network communication and mobile devices. In his bachelor studies he was focusing on hardware design and network communications. His first experience with computers was in 1993. He has 12 years of professional experience in information technologies.

### Personal information

Name: Jan Kubový  
Date of birth: December 12, 1984  
Place of birth: Prague, Czech Republic  
Nationality: Czech  
Academic degree: Ing., M.Sc.

### Contact

Email: [jan@kubovy.eu](mailto:jan@kubovy.eu)  
LinkedIn: <http://www.linkedin.com/in/kubovy>  
Xing: [https://www.xing.com/profile/Jan\\_Kubovy](https://www.xing.com/profile/Jan_Kubovy)  
GTalk: [jan@kubovy.cz](mailto:jan@kubovy.cz)  
Skype: [kubovyjan](https://www.skype.com/people/kubovyjan)  
Address: Softwarepark 23, Hagenberg im Mühlkreis, A-4232 Austria

## Specialities

- Programming languages: Java SE/EE, Ruby, PHP, C/C++, Prolog
- Databases: MySQL, PostgreSQL (pl/pgSQL), SQLite, Oracle
- Mobile devices: Java ME, Android
- Webdesign: HTML, JavaScript (AJAX), CSS, Flash
- Matlab, FriCAS, ASM, Informatica, ETL, UML, BPMN, XML, SOAP, GENA, UPnP, Latex
- Eclipse IDE, NetBeans IDE, Netbeans Platform
- OSes: (SE)Linux (RedHat, Fedora, Debian, Gentoo, Sabayon, Ubuntu), Mac OSX, Windows

## Education

### 2011 – present (Ph.D. expected)

**Johannes Kepler University in Linz (JKU)** - The Faculty of Engineering and Natural Sciences (TNF)

- At the **Institute for Application Oriented Knowledge Processing (FAW)**
- Main field of study: Informatics and Computer Science
- Publications:
  - Kubový, J., Auer, D., Küng, J.: Behavior-Based Decomposition of BPMN 2.0 Control Flow. In: ICEIS 2014 - Proceedings of the 16th International Conference on Enterprise Information Systems. p. 9. Institute for Systems and Technologies of Information, Control and Communication (INSTICC), SciTePress, Lisabon, Portugal (apr 2014) (to be published).
  - Auer, D., Hinterholzer, S., Kubový, J., Küng, J.: Business Process Management for Knowledge Work: Considerations on Current Needs, Basic Concepts and Models. In: F. Piazzolo, M. Felderer (eds.). Innovation and Future of Enterprise Systems - Proceedings ERP Future 2013 (Revised Papers), Lecture Notes in Information Systems and Organisation, Vol. 5, Springer (2014) (to be published).
  - Kubový, J., Küng, J.: Renement of BPMN 2.0 Inclusive and Complex Gateway Activation Concept towards Process Engine. In: F. Piazzolo, M. Felderer (eds.). Innovation and Future of Enterprise Systems - Proceedings ERP Future 2013 (Revised Papers), Lecture Notes in Information Systems and Organisation, Vol. 5, Springer (2014) (to be published).
  - Kubovy, J., Rady, M., Auer, D., Küng, J.: Abstraction Levels in the Abstract State Machine Method for System Specication. In: ACOMP 2013. pp. 63-70 (oct 2013).
  - Kubový, J., Auer, D., Küng, J., Rady, M.: Transition between Different Abstraction Levels in an Abstract State Machine Ground Model. In: Morvan, F., Tjoa, A.M., Wagner, R. (eds.) 24rd International Workshop on Database and Expert Systems Applications, pp. 227-230, IEEE Computer Society, Prague, Czech Republic (sep 2013).

- Kubový, J., Küng, J.: Notification Concept for BPMN Workow Interpreter Using the ASM Method. In: Moreno-Díaz, R., Pichler, F., Quesada-Arencibia, A. (eds.) Computer Aided Systems Theory - EUROCAST 2013, Lecture Notes in Computer Science, vol. 8111, pp. 452-459. Springer Berlin Heidelberg (2013).
- Kubový, J., Geist, V., Kossak, E.: A Formal Description of the ITIL Change Management Process Using Abstract State Machines. In: Hameurlain, A.; Tjoa, A.M., Wagner R. (eds.) 23rd International Workshop on Database and Expert Systems Applications, pp. 65-69, IEEE Computer Society, Vienna, Austria (sep 2012).

## **2010 – 2011 (M.Sc.)**

**Johannes Kepler University in Linz (JKU)** - The Faculty of Engineering and Natural Sciences (TNF)

- Main field of study: Informatics: Engineering and Management
- Research project for **Quanmax AG** company
- Double degree scholarship - one of two students from Open Informatics Programme on **Czech Technical University (CTU)**, Faculty of Electrical Engineering (FEE) awarded a double-degree scholarship based on academic results and interview
- Language: English, German

## **2009 – 2011 (Ing. equivalent to M.Sc.)**

**Czech Technical University in Prague (CTU)** - Faculty of Electrical Engineering (FEE)

- Main field of study: Open Informatics - Computer Engineering
- Minor field of study: Open Informatics - Software Engineering
- Winner of Google Developer Competition 2010

## **2005 – 2009 (Bc.)**

**Czech Technical University in Prague (CTU)** - Faculty of Electrical Engineering (FEE)

- Main field of study: Electrical Engineering and Informatics - Electronics and Telecommunication Engineering
- Bachelor Thesis: Real-time EEG network transmission, Department of circuit theory, FPGA Laboratory, June 2009
- Publications:
  - Št'astný, J., Doležal, J., Kubový, J., Černý, V.: Design of a modular brain-computer interface. In Applied Electronics, Applied Electronics, pp 319-322, September 2010. ISBN 978-80-7043-865-7

## Experience

### 2011 – present

Research and Development at **Institute for Application Oriented Knowledge Processing (FAW)** at Johannes Kepler University in Linz

*research in the field of business process modeling and formal methods, software development*

### 2010

Consultant/developer internship at **Cognitive Security s.r.o.** company<sup>1</sup>.

*experience with security analysis, network security, software development*

### 2009

Ruby developer at **Singularis s.r.o.** company

*software development*

### 2007 – 2008

Developer at **LMC s.r.o.** company (jobs.cz, prace.cz, tobjobs.sk, praca.sk)

*experience with managing project, specifying workflow, coordinating, software development*

### 2005

**24/7 Zone Inc.** in New York City - web and MIDP interface for mobile devices for CCTV systems

*software development for mobile devices*

### 2004 – 2010

#### Freelancer

*software development*

### 2002 – 2004

Web designer, graphic designer at **Kompakt s.r.o.** company

---

<sup>1</sup>A security software startup by the Czech Technical University in Prague (<http://www.cognitivesecurity.cz>)

## **Other**

### **Language abilities**

- **English (4)** - fluently, master and Ph.D. studies in English
- **German (4)** - fluently, two-year stay in Germany, now living in Austria
- **Czech (5)** - native

### **Interests**

- Driving license (A,B)
- International yacht driving license type C
- Active member of Rotaract Club Linz, RI District 1920
- Skiing, yachting, squash, soccer, tennis, bicycling, music, piano, travelling (Europe, Algeria, Egypt), etc.