Methodology for the derivation of product behaviour in a Software Product Line

Paul Istoan

▶ **To cite this version:**

Paul Istoan. Methodology for the derivation of product behaviour in a Software Product Line. Other [cs.OH]. Université de Rennes; Université du Luxembourg, 2013. English. NNT : 2013REN1S013 . tel-00925479

HAL Id: tel-00925479
https://theses.hal.science/tel-00925479

Submitted on 8 Jan 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

ANNÉE 2013

UNIVERSITÉ DE RENNES 1

UNIVERSITÉ DU LUXEMBOURG

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : INFORMATIQUE

Ecole doctorale Matisse

présentée par

Paul Alexandru ISTOAN

Methodology for the derivation of product behaviour in a Software Product Line

Thèse soutenue à Luxembourg
le 21-02-2013

devant le jury composé de :

Philippe LAHIRE
Prof. Dr. , University of Nice / rapporteur

David BENAVIDES
Prof. Dr. , University of Seville / rapporteur

Pascal BOUVRY
Prof. Dr. , University of Luxembourg / examinateur

Olivier BARAIS
Assoc. Prof. Dr. , University of Rennes 1 / examinateur

Jean-Marc JÉZÉQUEL
Prof. Dr. , University of Rennes 1 / directeur de thèse

Nicolas GUELFI
Prof. Dr. , University of Luxembourg / co-directeur de thèse

Nicolas BIRI
Dr. , Centre de Recherche Public Gabriel Lippmann / membre invite

Alfredo CAPOZUCCA
Dr. , University of Luxembourg / membre invite

To Ioana, Matei, Smaranda and Ioan.

CONTENTS

Page

Contents . . . 6

List of Figures . . . 10

List of Tables . . . 11

1. Introduction . . . 1

1.1 Research domain . . . 1

1.1.1 Software Engineering . . . 1

1.1.2 Software Product Lines . . . 2

1.1.3 Model Driven Engineering . . . 3

1.1.4 Business Process Modelling . . . 4

1.2 Problem statement . . . 4

1.3 Contributions of the thesis . . . 6

1.4 Thesis organization . . . 10

2. Background . . . 12

2.1 Software Product Lines . . . 12

2.1.1 General notions . . . 13

2.1.2 SPLE process . . . 14

2.1.3 Domain Engineering . . . 15

2.1.4 Application Engineering . . . 16

2.1.5 Benefits and disadvantages . . . 17

2.1.6 Variability in SPL . . . 19

2.1.7 Feature Modelling . . . 25

2.2 Model Driven Engineering . . . 29

2.2.1 Models and meta-models . . . 30

2.2.2 Model transformations . . . 33
2.2.3 Model driven language engineering . . . 37
2.3 Business processes . . . 40
2.3.1 Business process management . . . 42
2.3.2 Business process modelling . . . 44
2.3.3 Business Process Modeling Notation . . . 46
2.3.4 Petri Nets . . . 52

3. SPL methodology for the derivation of product behaviour . . . 60
3.1 Overview of the methodology . . . 62
3.2 Construction of the feature diagram . . . 67
3.2.1 Feature diagram dialect and meta-model . . . 67
3.2.2 Feature diagram construction process . . . 70
3.3 Creation of business process fragments . . . 74
3.3.1 Overview of business process fragments . . . 74
3.3.2 Business process fragment construction process . . . 76
3.4 Verification of business process fragments . . . 78
3.4.1 Verification of structural and behavioural correctness . . . 79
3.4.2 Business process fragment verification process . . . 81
3.5 Association of business process fragments to features . . . 83
3.6 Configuration of the feature diagram . . . 84
3.6.1 What is a feature diagram configuration? . . . 84
3.6.2 Feature diagram configuration process . . . 87
3.7 Product derivation specification . . . 89
3.7.1 Composition interfaces . . . 89
3.7.2 Composition operators . . . 90
3.7.3 Product derivation specification process . . . 92

4. Language for modelling and composing business process fragments . . . 96
4.1 What is a composable business process fragment? . . . 97
4.2 Abstract syntax . . . 100
4.2.1 Relation with BPMN standard . . . 100
4.2.2 Language meta-model . . . 102
4.2.3 Language support for composing business process fragments . . . 113

4.2.4 Language support for product derivation specification 123
4.3 Concrete graphical syntax 126
4.3.1 Direct definition of graphical concrete syntax 126
4.3.2 Meta-model based graphical concrete syntax 131
4.4 Translational semantics 139
4.4.1 Meta-model of Hierarchical Coloured Petri Nets 141
4.4.2 Model-to-model transformation from CBPF to HCPN 142

5. Verification of business process fragment correctness 155
5.1 Notion of *"correctness"* for business process fragments 156
5.2 Verification of structural correctness of business process fragments 158
5.3 Verification of behavioural correctness of business process fragments 163
5.3.1 Using HCPN for business process fragment verification 164
5.3.2 Verification of general behavioural properties 165
5.3.3 Verification of fragment specific behavioural properties 169

6. Exemplification of the proposed methodology and tool support 175
6.1 Introducing the bCMS case study 175
6.2 Applying the proposed SPL methodology on the bCMS case study 178
6.2.1 Construction of the feature diagram 178
6.2.2 Creation of business process fragments 183
6.2.3 Verification of business process fragments 190
6.2.4 Association of business process fragments to features 194
6.2.5 Configuration of the feature diagram 196
6.2.6 Product derivation specification 199
6.3 Tool support 206
6.3.1 Tool requirements 206
6.3.2 General architecture of the tool 207
6.3.3 Modules of the tool 212

7. Perspectives 226
7.1 Defining composition operators for the CBPF language 226
7.1.1 Mathematical specification of business process fragments 228
7.1.2 Proposed composition operators 230
7.2 Composing business process fragments using aspect weaving 245
7.3 Modelling data for business process fragments 252

8. Conclusion 254

Appendix 273

A. Annex 2: business process fragments for the bCMS case study 275

LIST OF FIGURES

2.1 General SPL engineering process [PBvdL05] 15
2.2 Classification of SPL variability modelling approaches. 22
2.3 Feature diagram dialects - synthesis of variability modelling concepts 27
2.4 The four layer meta-modeling architecture 33
2.5 Overview of Model Transformation process 35
2.6 Graphical notation for core set of BPMN elements 52

3.1 General steps of the proposed methodology 66
3.2 Feature diagram meta-model 68
3.3 Complete process for creating the feature diagram 71
3.4 Complete process for creating business process fragments 77
3.5 Complete process of verification of business process fragments 82
3.6 Feature diagram meta-model: associating fragments to features 83
3.7 Complete process of associating business process fragments to features from the FD 85
3.8 Complete process of configuring the feature diagram 88
3.9 Steps of the product derivation specification process 93

4.1 Frequency distribution of BPMN construct usage [Rec10] 101
4.2 Core structure of the business process fragment modelling language 103
4.3 Excerpt of composable business process fragment meta-model: flow objects 105
4.4 Excerpt of language meta-model: swimlanes and connecting objects 106
4.5 Excerpt of language meta-model: newly introduced concepts 110
4.6 Excerpt of language meta-model: composition interface 111
4.7 Meta-model of composable business process fragment modelling language . . 112
4.8 Excerpt of language meta-model: composition operators 122
4.9 Excerpt of language meta-model: support for product derivation specification 125
4.10 Graphical concrete syntax: representation of activities, events and gateways 128

4.11 Graphical concrete syntax for: swimlanes, connecting objects and artifacts . 129
4.12 Graphical concrete syntax for product derivation specification elements . . . 130
4.13 A general model of diagrams for describing the concrete syntax of the language 132
4.14 Concrete graphical syntax and relation with abstract syntax - part 1 135
4.15 Concrete graphical syntax and relation with abstract syntax - part 2 137
4.16 Example of business process fragment created with CBPF : transportation booking 138
4.17 Meta-model of Hierarchical Coloured Petri Nets 140
4.18 Mapping template from CBPF to HCPN: task and sub-process 143
4.19 Mapping template from CBPF to HCPN: gateways 145
4.20 Mapping template from CBPF to HCPN: events 147
4.21 Mapping template from CBPF to HCPN: sequence flow, message flow, data association, data objects and composition tags 149
4.22 Mapping transportation reservation fragment to HCPN: Step 1 151
4.23 Mapping transportation reservation fragment to HCPN: Step 2 153

5.1 Generic query function for node search and processing 173
5.2 Pseudo-code description of *SearchNodes* function 174

6.1 Overall view of the environment and the desired system 176
6.2 Domain model of the bCMS system 179
6.3 Complete feature diagram of the bCMS system 182
6.4 Communication establishment business process fragment 184
6.5 Creation of coordinated route plan business process fragment 186
6.6 Closing the crisis business process fragment 187
6.7 PSC send and receive business process fragment 188
6.8 Multiple crisis business process fragment 188
6.9 SSL communication protocol for vehicle management business process fragment 189
6.10 PSC authentication using symmetric encryption business process fragment . 191
6.11 Transforming the *Creation of coordinated route plan* business process fragment into a HCPN 192
6.12 Connecting features to business process fragments for the bCMS case study 195
6.13 Feature diagram configuration of bCMS product 198
6.14 "Coordinator identification" business process fragment after adding composition tags 201

6.15 "Objective complete coordination" business process fragment after adding composition tags . . . 202
6.16 First part of composition workflow for bCMS example . . . 204
6.17 Second part of composition workflow for bCMS example . . . 205
6.18 General architecture of SPLIT tool . . . 208
6.19 Usage of the SPLIT tool . . . 211
6.20 Screenshot of the Feature Diagram Editor . . . 213
6.21 CBPF to HCPN tranformation using ATL: mapping the root elements . . . 217
6.22 CBPF to HCPN tranformation using ATL: mapping CBPB objects into HCPN elements . . . 217
6.23 Screen-shot of CPN Tools interface . . . 218
6.24 Architecture of the *Fragment to aspect adapter* module . . . 222
6.25 Join-Point meta-model . . . 223
6.26 Class Diagram of the Join Point Detector Module . . . 224
6.27 Class Diagram of the Weaver Module . . . 225

7.1 Sequential composition operator for business process fragments . . . 231
7.2 Parallel composition operator for business process fragments . . . 232
7.3 Exclusive choice composition operator for business process fragments . . . 234
7.4 Choice composition operator for business process fragments . . . 235
7.5 Unordered (arbitrary) sequence composition operator for business process fragments . . . 237
7.6 Parallel with communication composition operator for business process fragments . . . 239
7.7 Refinement composition operator for business process fragments . . . 240
7.8 Synchronization composition operator for business process fragments . . . 242
7.9 Insert after composition operator for business process fragments . . . 244
7.10 Sequential weaving with aspects . . . 246
7.11 Choice composition of business process fragments . . . 247
7.12 Initial fragment and choice operator . . . 248
7.13 Pointcut and Advice built for Choice composition . . . 248
7.14 Exclusive choice composition of processes with aspects . . . 249
7.15 The eight aspects for the choice composition . . . 250
7.16 Input of the refinement composition . . . 251
7.17 Pointcut1 and Advice built for Refinement composition . . . 252

A.1 Communication establishment business process fragment 276
A.2 Coordinator identification business process fragment 276
A.3 Crisis details exchange business process fragment 277
A.4 Creation of coordinated route plan business process fragment 277
A.5 Vehicle dispatch coordination business process fragment 278
A.6 Vehicle target arrival coordination business process fragment 278
A.7 Crisis objective complete business process fragment 279
A.8 Vehicle return coordination business process fragment 279
A.9 Close crisis business process fragment 280
A.10 Vehicle management - PSC send and receive business process fragment . . . 280
A.11 Vehicle management - FSC send and receive business process fragment . . . 281
A.12 Vehicle management - PSC receive business process fragment 281
A.13 Multiple crisis business process fragment 282
A.14 SOAP communication protocol business process fragment 282
A.15 SSL communication protocol business process fragment 283
A.16 Encrypted data communication business process fragment 283
A.17 Password based authentication business process fragment 284
A.18 Certificate based authentication business process fragment 284
A.19 Symmetric encryption authentication business process fragment 285
A.20 One time password based authentication business process fragment 286
A.21 Authentication based on mutual authorization business process fragment . . 287
A.22 HTTP based communication layer business process fragment 288
A.23 SOAP based communication layer business process fragment 288

LIST OF TABLES

1. INTRODUCTION

Abstract

In this chapter, both the problems that address this thesis and the objectives that motivated its development are introduced. The chapter opens with the presentation of the dimensions of interest that represent the research domain. Then, the current problems that have been found in this domain are clearly stated. This is followed by a list of contributions of this thesis, aimed at solving these identified problems. The chapter ends with an overall description of the organisation of the thesis.

1.1 Research domain

1.1.1 Software Engineering

The increasingly complex and competitive market situation places intense demands on companies, requiring them to respond to customer needs, and to deliver more functionality and higher quality software faster. Software industry is constantly facing increasing demands for "better, faster, cheaper" and the increasing complexity of software products and projects have significantly "raised the bar" for software developers and managers to improve performance.

Ed Yourdon in his foreword for the "Managing Software Requirements" book [LW99], describes software systems as being, by their nature, *"intangible, abstract, complex, and â in theory at least â "soft" and infinitely changeable"*. All these indicate that software development is a highly complex, dynamic task, which is not only determined by the choice of the right technologies, but also â to a large extent â by the knowledge and skills of the people involved. The success of software organisations depends on their ability to facilitate continuous improvement of products and on the effectiveness and efficiency of software product development.

The significant impact of software on today's economy generates considerable interest in making software development more cost effective and producing higher quality software. However, almost since software engineering emerged, software engineers have had to cope with the famous "software crisis", challenging their abilities to provide satisfactory solutions within a reasonable time. All over the world, organizations developing software intensive systems are today faced with a number challenges. These challenges, related to characteristics of both the market and the system domain, may include:

- Systems grow ever more complex, consisting of tightly integrated mechanical, electrical/electronic and software components.

- Systems are often developed in series, ranging from a few to thousands of units. This implies that it is important to achieve efficient development, since development costs are carried by only a few units.
- Systems have very long life spans, typically 30 years or longer. This implies that it is important to develop high quality systems, and to achieve effective maintenance of these systems once developed.
- Systems are developed with high commonality between different customers; however systems are always customized for specific needs. This implies that there is potential for high levels of reuse of development efforts between different customer projects.

Throughout the last decades, the software engineering community has developed some innovations that are enabling engineers to tame the inherent complexity of modern systems and to develop them more rapidly. Among them, this thesis identifies *software product lines*, *model driven engineering* and *business process modelling* as mainstays to define an approach capable to address the aforementioned challenges.

1.1.2 Software Product Lines

Software reuse has become a significant ingredient of software development due to rapid and large amount of software production. The organizations adopt this approach to reduce cost, time to market and to increase the quality of the software. During 1980âs object oriented programming brought reuse in the form of classes and later, component based software development introduced reuse in the form of components. These methods did not obtain the original benefits of reuse due to the usage at a very small scale and opportunistic reuse.

In the late 1980s a growing number of software development organizations started adopting approaches that emphasize proactive reuse, interchangeable components, and multi-product planning cycles to construct high-quality products faster and cheaper. Standard methods, referred to as *software product line* or *software family* practices, have developed around these approaches.

The concept of *"product line"* is not new and engineers in various domains, such as the automotive sector, have adopted this concept of development for the last few decades, to benefit from the advantages that SPL engineering offers. However, with regard to software, systematic reuse, including variability concepts, is still challenging and a relatively new problem.

Software product line engineering (SPLE) came into being in 1990s. In this approach the reuse is planned, large in scale, wide in range and profitable. The reuse includes artefacts which are costly to develop from scratch and planned in such a way to be used in a family of related products. SPLE is an extensive approach to organizing the continuous development of software products, where the main property is the planned, prepared, and anticipated reuse of a set of domain artefacts for later fast and efficient composition of applications. This software development paradigm enables reuse of common parts but at the same time allows for variations. The basic idea of this approach is to use domain knowledge to identify common parts within a family of related products and to separate them from the differences between the products. The commonalties are then used to create a product platform that can be used as a common baseline for all products within such a product family.

SPLs are gaining widespread acceptance and various domains already apply SPL engineering successfully to address the well-known needs of the software engineering community, including increased quality, saving costs for development and maintenance, and decreasing time-to-market. SPLs offer a systematic reuse of software artefacts within a range of products sharing a common set of units of functionality.

This thesis will analyse the SPL engineering domain and its latest progress. It will identify some of the issues that are currently being faced for applying software product line engineering approaches and will propose viable solutions to those problems.

1.1.3 Model Driven Engineering

As we have seen, the main problem software engineers are faced with is complexity. It may be accidental (i.e. related to a particular technology we are using to develop systems) or essential (related to the problem to solve). One possible approach to deal with complexity is the use of *models*. Models have been used in various engineering disciplines such as civil engineering or anatomy to reason about the system to be built.

Modelling is a cornerstone of all traditional engineering disciplines [Sel03]. From the conception and design, through the construction and maintenance of any engineered system, modelling plays a crucial role. Throughout the last decades software engineers have explored how lessons learned from traditional engineering disciplines can be applied to the design, construction, deployment and maintenance of software systems. The solution proposed is called Model Driven Engineering (MDE). This software development paradigm raises the abstraction level for system specification and is highly regarded as a viable solution for building complex software systems. The overall objective of MDE is to increase productivity and reduce the time to market by enabling the development of complex systems by means of models [Sel03].

The guiding principle of this new software engineering trend is to focus on models rather than on computer programs. Models provide a view of the system from a certain perspective, that is they concentrate on some relevant aspects of the system while abstracting others. Therefore, models are easier to read and facilitate the understanding of the system. Unlike in civil engineering, software models can also be used to actually build the system by transforming them from requirements to system implementation.

MD also promotes the notion of *"platform independence"*, as a solution to perpetuate software design decisions with respect to technological changes. Abstract models of a system can be freed from any information about the technology that will be actually used to develop the system. Technology-dependent concrete models can be generated thanks to model transformation and via separate models describing the target platform.

This thesis adheres to the MDE principles, in particular for domain specific language design. We use the meta-modelling technique when addressing the definition of modelling languages (for abstract and concrete syntax definition). Moreover, the concept of model transformation is also extensively used. Hence, the notions of model, meta-model, model conformance and model transformation are major concerns on which this thesis relies on for the achievement of its contributions.

1.1.4 Business Process Modelling

We have seen that the main focus of the business and commercial worlds is to automate and improve production efficiency, reduce costs and tame the complexity of modern systems. Thus, many companies had to improve their business to keep their customers. This moment triggered the awareness of organisations of the importance of *business processes*. They became the key to a successful business.

Value-adding processes have become more and more the principle of organising the business. Hence the modelling of business processes is becoming increasingly popular. Both experts in the field of Information Technology and Business Engineering have concluded that successful systems start with an understanding of the business processes of an organisation.

A business process is the combination of a set of activities within an enterprise with a structure describing their logical order and dependence whose objective is to produce a desired result. Business process modelling enables a common understanding and analysis of a business process. A process model can provide a comprehensive understanding of a process. An enterprise can be analysed and integrated through its business processes. Hence the importance of correctly modelling its business processes.

Business Process Management (BPM) is an established discipline for building, maintaining, and evolving large enterprise systems on the basis of business process models. Organizations attempt to improve their business performance by applying BPM methods. BPM has become an essential way of controlling and governing business processes. Business process modelling is a key phase of the BPM life-cycle, which intends to separate process logic from application logic, such that the underlying business process can be automated [SBW04]. The modelling of business processes is becoming increasingly popular and plays a pivotal role in the business process management discipline.

However, the modelling of business processes is complex due to several reasons. The real-world processes are large and complex but must be captured in process models. This thesis makes use of business processes and business process modelling to a large extent. There is a clear synergy between business process modelling and MDE which we try to exploit. Moreover, all of these concepts are applied in the context of software product line engineering.

1.2 Problem statement

Software Product Line Engineering (SPLE) is a recent software development paradigm offering software suppliers/vendors new ways to exploit the existing commonalities in their software products and to support a high level of reuse, thus generating important quantitative and qualitative gains in terms of productivity, time to market, product quality and customer satisfaction. This technique has gained a lot of attention in recent years by both research and industry. The SPLE process consists of two major steps: Domain Engineering deals with core assets development, while Application Engineering addresses the development of the final products.

Throughout the past years, the product line community has mainly focused on the Domain Engineering phase of the process. A review of SPLE literature indicates that Application Engineering (product derivation), a key phase of the SPL process that can be tedious and

error-prone, has been given far less attention compared to Domain Engineering. Implicitly, there arises the need for new product derivation techniques in the SPL research field.

To address this situation, SPLE has recently turned towards Model-Driven Engineering (MDE), identified as a software development paradigm able to offer viable solutions for improving product derivation. MDE advocates the use of models to face the inherent complexity of software systems. The result of the derivation process is the model of an individual product obtained from the core assets. Two types of models, each offering a different view of the derived product, can be obtained: *structural* and *behavioural*. Structural models provide a static view of the derived product. Behavioural models illustrate the dynamic behaviour of the product and the general flow of control. Most of the work in SPLE addresses the derivation of structural product representations, neglecting or just briefly addressing the problems inherent to the derivation of product behaviour. This yields an unwanted situation, as the behavioural product representation is as important as the structural one. The few existing techniques that try to address to some extent the issue of deriving product behaviour lack the "end-to-end" dimension, meaning they do not cover both domain engineering and application engineering phases of the SPLE process.

Taking into consideration all of the afore mentioned reasons, the major problem addressed in this thesis is *the definition of a methodology for software product line engineering that covers both Domain Engineering and Application Engineering phases of the SPLE process and which focuses on the derivation of behavioural models of SPL products.* By applying this methodology we want to produce behavioural models that belong to the analysis and early design levels of the software development life-cycle. Thus, the behavioural models obtained as a result of applying this methodology should describe the business and operational step-by-step workflows of activities/actions performed by the derived product. BPMN, the standard for modelling business processes, has been selected as the specific type of model used for representing the behaviour of derived products. The proposed methodology can be applied in general software engineering domains, it does not target a specific one. We require several qualities from the proposed methodology: scalable, comprehensible, suitable, expressive enough, can be easily maintained and supports modifications.

From another perspective, we want to develop this methodology following model driven engineering principles. This is due to the fact that in software engineering, models allow to express both problems and solutions at a higher abstraction level than code. MDE treats models as first-class elements in the software development process. By applying an MDE approach, it is possible to reduce design complexity and make software engineering more efficient by shifting the focus from implementation to modelling.

Another challenge that lies ahead is to propose and deliver the appropriate tool support for the methodology. The availability of good tool support will enable users to better understand and more easily apply the proposed methodology. Moreover, tool support facilitates assessing the entire methodology on appropriate case studies for showcasing its characteristics and strong points.

As the main focus of the methodology is to obtain behavioural representations of SPL products, this also implies to solve the following problem: how to model a complex behaviour starting from several simpler ones? One of the factors that contributes to the difficulty of developing complex behaviours is the need to address multiple concerns in the same artefact. This situation emphasizes the need for separation of concerns (SOC) mechanisms as a support to the modelling of complex behaviours, represented using business processes in this case: concerns are defined separately, and assembled into a final

system using composition techniques. This challenge is pointed out by Mosser in [Mos10]: *"there is no approach described in the literature which fulfils the specific goal of supporting the modelling of complex business processes following a compositional approach, at model level"*.

Another challenge lies in finding both the adequate behavioural formalism that fits the needs of the analyst as well as a formal composition mechanism that facilitates the generation of the expected behavioural model.

Regarding the actual composition of business processes, there are currently only a few proposals. This is currently very much a manual activity, which requires specific knowledge in advance and takes up much time and effort. The composition problem is one that cannot easily be solved by a "copy and paste" approach, as it may introduce problems like: redundancy, update anomalies or inconsistent behaviour in the resulting models. There is also a need for a formal foundation and notation for the compositions which allows the creation of business process models from model fragments. All the afore-mentioned reasons make the question: *"how to build a complex behaviour based on simpler ones?"* complex to answer.

1.3 Contributions of the thesis

The main contributions of this thesis are the following:

- **A notion of *composable business process fragment* with the appropriate language support**

  A major contribution of this thesis is to introduce the notion of *composable business process fragment* as a new unit of reuse for business process modelling. Semantically, a business process fragment specifies a single abstract functionality. It implements the behaviour of a single feature. Business process fragments are blocks of process logic with strictly defined boundaries. They fulfil the need of another unit of reuse, one that allows fine-grained reuse of process logic within the range from atomic language constructs to sub-processes and whole processes. Fragments have relaxed completeness and consistency criteria compared to regular business processes and may be partially undefined.

  We also propose a language that supports the modelling of such *composable business process fragments*. It is called CBPF and is based on the BPMN language, the standard defined by OMG for modelling business process flows which promotes a process oriented approach for modelling system behaviour. We use BPMN as the basis for defining the CBPF language because it provides a notation that is easily readable, usable and understandable by both technical and business users. Based on existing studies [zMR08], our language contains only those BPMN elements proven to be essential for modelling behaviour.

  To address the issue of process composition, the CBPF language proposes the concept of *composition interface* and *composition tag*. Using an annotation-based mechanism, composition interfaces are used to explicitly identify the parts of a process fragment where it can connect to other fragments or where other fragments can be connected to it. The interfaces are also an indicator of how this connection can be performed.

The CBPF language is created following model driven engineering principles. The abstract syntax of the language is specified by defining its meta-model which represents in an abstract way the concepts and constructs of the modelling language, and providing the means to distinguish between valid and invalid models. A set of consistency rules, described using the Object Constraint Language (OCL), are added on the meta-model in order to express well-formedness constraints for business process fragments. We then propose a unique graphical concrete syntax for the language. It is a crucial element of language design and we therefore treat it as a separate element within the language description. It is based on the graphical syntax proposed by BPMN, enriched by concrete syntax representations for the newly introduced concepts. Finally, we define the semantics of the CBPF language following a translational approach.

- **A model-to-model transformation that automates the mapping from the CBPF language towards the Petri Net formalism**

  Following an MDE approach, we propose a model-to-model transformation that translates the newly proposed business process fragment modelling language into the Petri Net formalism. We use a particular class of Petri Nets, called hierarchical coloured Petri Nets, that have several properties useful when creating a proper mapping. This model transformation is used for defining the semantics of the CBPF language in a translational manner.

  Model-to-model transformations are used to normalize, weave, optimize, simulate and re-factor models, as well as to translate between modelling languages. This is the type of model transformation that we propose. We specify it in terms of a mappings of elements from the business process fragment modelling language to equivalent constructs in Hierarchical Coloured Petri Nets (HCPN). The transformation is unidirectional and covers the entire set of elements.

  The goal of this transformation is twofold:

  - as the business process fragment modelling language does not have a formally defined semantics, the mapping is used for provide a formal semantics to the language using a translational semantics;
  - as Petri Nets are a well-known formalism, several well defined analysis approaches and tools exist for it. The second goal of the mapping is to allow access to these tools and thus facilitate the verification of behavioural correctness of business process fragments.

- **A new set of composition operators created specifically for the composition of CBPF models**

  An essential part of any composition process are the composition operator that are applied. Another contribution of this thesis is to propose a set of composition operators created specifically for composing business process fragments. The operators are inspired from well known and well defined composition operators proposed for the Petri Net language.

  We formally define a set of 10 binary composition operators for business processes. The operators are included in the business process fragment meta-model. The semantics of each operator is defined using a translational semantics towards an equivalent Petri net composition operator. The notion of composition interface is crucial for the

specification of the operators, as it indicates precisely the places where the models taken as input by the operator will be modified during the actual composition.

These composition operators are used during the product derivation process, for composing business process fragments into the final product behaviour. However, they are valuable by themselves and can also be used independently from the SPL context, whenever we need to compose two business processes into a new one.

- **An approach for verifying the *correctness* of business process fragments from both a structural and behavioural perspective**

  The notion of *correctness* for composable business process fragments is defined as the summation of two simpler properties: *structural correctness* and *behavioural correctness*.

  - *Structural correctness:* a business process fragments is considered to be structurally correct if it satisfies a set of *consistency rules*. We propose a set of such well-formedness rules for business process fragments and specify them using OCL directly on the CBPF meta-model. Therefore, all business process fragments created based on this meta-model will have to satisfy these consistency rules to be valid with respect to the CBPF meta-model.
  - *Behavioural correctness:* requires the verification of several dynamic properties on a business process fragment, which cannot be statically checked. There are two types of behavioural properties that we want to verify. First of all, we propose to check a predefined set of general properties, which should hold on every business process fragment, like: absence of dead states, live-lock analysis, reachability analysis for end states and composition interfaces. Secondly, we want to allow the user the possibility to verify certain properties specific to individual fragments. To enable this, we provide several generic verification templates written in the CPN query language which the user can adapt to his particular needs. For performing all the behavioural verifications, we first apply a model-to-model transformation that transforms the business process fragment into an equivalent hierarchical coloured Petri net. The verification of the behavioural properties is then performed on the resulting Petri net using CPN Tools, a well known Petri net verification tool.

- **A new SPLE methodology, focusing on the derivation of product behaviour**

  Another main contribution of this thesis is a new software product line engineering methodology that focuses on the derivation of product behaviour. By applying this methodology, we can produce behavioural models that belong to the analysis and early design levels of the software system development life-cycle. The proposed methodology covers only the derivation of behavioural product models and does not address the structural product representation. However, it can be used together with other product derivation techniques for obtaining the structural product models.

  The methodology follows the classical SPLE process [vdL02] and covers both Domain Engineering and the Application engineering phases:

  - during Domain Engineering, we propose to capture domain knowledge using the newly introduced concept of *reusable/composable business process fragments*. These composable process fragments represent our core assets base, from which

new behavioural product models will be later created. We choose to capture the commonality and variability in the domain in a separate variability model, represented as a feature diagram. We apply the SOC principle and keep the core assets and the variability representations separate. Moreover, in order to facilitate the product derivation process, we connect features from the feature diagram to business process fragments by association. This relation is explicitly specified in the feature diagram meta-model. Moreover, we want to ensure that business process fragments from the core assets base are correct prior to composition. Therefore, we propose to apply the approach for the verification of business process fragment correctness from structural and behavioural perspective which was previously mentioned.

– during Application engineering, we create new products from the core assets base using a compositional approach. We propose a new derivation approach that uses positive variability and which creates a new business process that models the behaviour of the derived product. In a first step, we require the contribution of the user for creating a particular product configuration based on a selection of features performed on the feature diagram. Once the selection is done, the business process fragments associated to the selected features are also implicitly selected. The next step is to create a composition workflow that explicitly defines both the order in which the selected fragments are composed and also the composition operators that will be applied. The composition process itself is influenced by the composition interfaces defined on the business process fragments. The composition operators that we previously proposed are used for composing the business process fragments, resulting the final behavioural product representation.

- **Tool support for the proposed SPLE methodology**

  Good tool support is one of the key elements for the fast adoption of any new methodology and language. Thus, it is of the utmost importance to provide the product line engineer with a tool that will allow him to practically apply the concepts and ideas proposed by our methodology. Moreover, after designing a domain-specific language like CBPF, the next important task is to determine how to provide the supporting tools for the modelling language. Thus, the SPLIT tool suite, a tool that supports the users in applying all the phases of our SPL methodology is also proposed in this thesis. We describe initially the general requirements that such a tool should fulfil, like: Support for modelling and configuring feature diagrams, creating CBPF models, support for verifying and composing business process fragments. The SPLIT tool suite has been developed as a set of Eclipse plug-ins which are meant to be integrated as a single tool that is capable of fulfilling the previously mentioned requirements. We propose a modular tool architecture, to facilitate plugin-development and development iterations, by distributing different tasks of the tool to different modules of the architecture. This allows the swapping of strategies and approaches, while maintaining a fully functional tool, and in this manner quantitatively and qualitatively compare multiple implementation options for the same module section.

  Finally, for better understanding, the methodology is applied on a medium scale case study. This serves to point out the characteristics of the approach and also its strong and weak points.

1.4 Thesis organization

The thesis is organised in eight chapters, plus some appendixes with additional information useful for the understanding of the thesis content. In the following, a summary of the content of each chapter and appendix is given.

Chapter 2 introduces the necessary background related to the areas of software product line engineering, model-driven engineering and business process modelling, as the claimed thesis contributions rely on these domains. Hence, it is in this chapter that the reader will be introduced to the concepts and principles that govern the field of Model-Driven Engineering. Core concepts like *model*, *meta-model* and *model transformation* are presented. Special attention is dedicated to the definition of domain specific languages using following an MDE approach. We also present software product line engineering, which is rapidly emerging as a viable and important software development paradigm aimed at handling the exponential increase in complexity and variability of modern software. Finally, we present business processes and business process modelling, since experts in the fields of Information Technology and Business Engineering have concluded that successful software systems start with an understanding of the business processes of an organisation.

Chapter 3 presents one of the main contributions of this thesis and proposes a new software product line engineering methodology that focuses on the derivation of product behaviour. A methodology can be seen as a framework for applying software engineering practices with the specific aim of providing the necessary means for developing software-intensive systems. By applying the proposed methodology, behavioural product models can be produced that belong to the analysis and early design levels of the software development life-cycle. The behavioural models obtained should describe the business and operational step-by-step workflows of activities/actions performed by the derived product. We first define the main flow of the methodology and then describe in detail its specific individual steps.

Chapter 4 presents a new domain specific language called CBPF created specifically for modelling composable business process fragments. The most common approach to obtain business process fragments is to create them from scratch, as concrete implementations of the features from the feature diagram of the SPL. For this purpose, adequate language support is required. We start by precisely defining what a business process fragment really is. Then, a model driven approach is followed for creating and specifying the CBPF domain specific language. We first describe the high-level structure of the CBPF language and its abstract syntax by means of a meta-model representation. We continue the language description by we proposing a unique graphical concrete syntax. We conclude the chapter by defining the semantics of the CBPF language following a translational approach, by proposing a mapping of CBPF concepts onto the Hierarchical Coloured Petri Net (HCPN) formalism.

Chapter 5 several types of verifications that can be applied to business process fragments in order to determine their *"correctness"*. Business process fragment verification is also a key step of the proposed SPL methodology. We first define the notion of *"correctness"* for business process fragments as the summation of two other properties: structural correctness and behavioural correctness. The structural verification of a business process fragment is ensured by defining a set of adequate fragment consistency rules that should be valid for every business process fragment that can be created with the CBPF language. These well-formedness rules are defined using OCL directly on the CBPF meta-model. To perform the verification of behavioural correctness, we must first transform the business

process fragment under analysis into an equivalent HCPN with the help of a model-to-model transformation that we propose. Business process fragment behavioural properties are separated into two major classes: *generic* ones which specify general dynamic properties that any business process fragment should fulfil; *fragment specific* properties for which we propose several property templates that can be adapted and used by the product line engineer to check them.

Chapter 6 exemplifies the proposed SPL methodology by applying it to a case study from the crisis management system domain. the case study also serves to facilitate the understanding of the concepts and functioning of the CBPF language, and also to exemplify the proposed verification techniques of business process fragments. After briefly introducing the bCMS case study, we follow the methodology and, for each of its steps, explain and exemplify how it applies on the case study. In the second part of the chapter we present the SPLIT tool suite, which is the tool support that we propose for our methodology. We describe the general requirements that such a tool should fulfil. We then present the general architecture of the proposed tool and discuss in more details the different tool modules and the functionalities each of them provides.

Chapter 7 describes extensions, improvements and potential directions for future research. We propose a set of composition operators for the CBPF language, designed specifically for composing business process fragments. We also propose to investigate the integration of data and data modelling for business process fragments. Furthermore, we identify aspect weaving as a possible approach for composing business process fragments, and present how two fragments together with the composition operator to be applied for composing them can be transformed into base and aspect and woven together.

Finally, Chapter 8 concludes the thesis, summarising the main achieved results.

2. BACKGROUND

Abstract

The research context of this thesis is scoped by three research areas: (1) Software Product Lines (SPL), (2) business processes, and (3) Model-Driven Engineering (MDE). The goal of this chapter is to introduce the relevant concepts belonging to each of these areas in order to facilitate the understanding of this thesis for the reader and allow him to acquire the required background on which the claimed thesis contributions rely on. The presentation of the background is structured in three parts, one for each area of interest. Section 2.1 introduces the general terminology about Software Product Lines and discusses what makes them successful. The general SPL engineering process is then presented. Individual sub-sections are dedicated to discussing in more details the characteristics of each step of the process. We then focus on variability, one of the key characteristics that distinguishes SPL from other software engineering approaches. A separate sub-section is dedicated to Feature Modelling, the most popular SPL variability modelling technique. Section 2.2 presents the MDE field by giving information about its aim, and describing in details the concepts and principles that govern this area. We discuss the importance of models and meta-models for MDE and the use of model transformations. A presentation of how MDE principles can be applied to language engineering is also provided. Finally, section 2.3 addresses business processes and business process modelling and their increasing importance in modern enterprises. BPMN, the standard for modelling business processes and process flows, is discussed in details. We end with a presentation of the formal language Petri Nets.

2.1 Software Product Lines

Until recently, software systems were either designed to have an extensive list of possible features, or they were particularly produced for a single customer. Furthermore, software solutions were originally quite static and every change implied extensive changes of existing source code. This is no longer an option for contemporary software systems. Software solutions also have to deal with an exponential increase in complexity and variability, due to the constant evolution of the market. Interests of the software producer, to maximize his benefits and minimize production costs, come into contradiction with those of the customer, who expects an increase in quality of the delivered software. Therefore, especially when size and complexity exceed the limits of what is currently feasible with traditional approaches, new approaches to software development to address the above mentioned issues are required by the software development community.

Software Product Lines (SPL), or *software families*, are rapidly emerging as a viable and important software development paradigm aimed at handling such issues [Nor99]. Use

of product line approaches allowed renowned companies like Hewlett-Packard, Nokia or Motorola to achieve considerable quantitative and qualitative gains in terms of productivity, time to market and customer satisfaction [SEI].

2.1.1 General notions

The *"software product line"* concept has its origins in the program families approach of Parnas [Par76]. It only drew the attention of the software engineering community when software began to be massively integrated in families of hardware products, cellular phones [MH05] being the best known example. Several other areas, like automotive systems, aerospace or telecommunication are also targeted by software product lines.

In the 1990's, industry was confronted with an increasing demand for individualised products, which meant taking into account the customers' requirements. For the companies, this implied higher investments and therefore lower profit margins. Many companies started to introduce *common platforms* for their different types of products and plan in advance which parts can be used in different products. By combining mass customisation and a common platform, a higher level of reuse and customer satisfaction can be achieved. The application of these principles for the development of software-intensive systems gave birth to the software product line engineering paradigm. The use of software product line engineering practices can efficiently satisfy this need for software mass customization [PBvdL05, Dav87].

Several definitions can be found in the research literature for the *"software product line"* concept:

- Clements et al. define the concept as follows: *"a software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way"* [CN01]. This definition captures the fundamental idea of SPL: to reuse a base of managed software artefacts to systematically define, design, build and maintain a set of related products in a given domain.
- Bosch defines the concept somewhat differently: *"A software product line consists of a product line architecture and a set of reusable components that are designed for incorporation into the product line architecture. In addition, the product line consists of the software products that are developed using the mentioned reusable assets"* [Bos00]. These two definitions share the notion of *set of reusable or core assets*. Nevertheless, they provide different perspectives of the concept: *market-driven*, as seen by Clements et al., and *technology-oriented* for Bosch.
- We also retain the definition of Pohl et al.: *"Software product line engineering is a paradigm to develop software applications (software-intensive systems and software products) using platforms and mass customisation"* [PBvdL05]. It focuses on the idea of software mass customization and the use of a common platform.

SPL engineering (SPLE) focuses on capturing the *commonality* and *variability* between several software products [CHW98]. Instead of describing a single software system, a SPL describes a set of products in the same domain. This is accomplished by distinguishing

between elements common to all SPL members, and those that may vary from one product to another. Reuse of *core assets*, which form the basis of the product line, is highly encouraged. These core assets extend beyond simple code reuse and may include the architecture, software components, domain models, requirements statements, documentation, test plans or test cases [ZJ06a].

The notions of *commonality* and *variability* are of the utmost importance when referring to software product lines. In this context, they can be defined as:

- *Commonality*: a property held uniformly across all the members of the SPL;
- *Variability*: a property about how members of a SPL differ from each other.

Reuse is another core concept for the SPL paradigm. Software reuse has long been regarded as the answer to the "software crisis" [Gib94]. The main goal of software reuse is to improve software quality and productivity, thereby maximizing a software development organization's profits [FK05]. At a first glance, SPL development might resemble to traditional software reuse, but it is actually much more elaborated. In product line development reuse is planed, enabled and enforced [SEI12]. The core assets are the reuse repository of a software product line. They include all the artefacts that are the most costly to develop: domain models, requirements, architecture, components, test cases, performance models, etc. Furthermore, these core assets are from the beginning developed to be (re)used in several products.

SPL focuses on strategic software reuse: "consolidate commonality throughout the product line, strategically manage all product line variation, and aggressively eliminate all duplication of engineering effort" [SPL]. Adopting a software product line approach requires an organization to move from developing single products to developing product families. This means that everything is developed with reuse in mind, so the effort needed to customize the reusable assets to fit a new system is largely reduced compared to traditional reuse approaches.

2.1.2 SPLE process

Adopting the SPLE paradigm implies performing two main activities: *domain engineering* and *application engineering* [WL99, vdL02].

The *domain engineering phase*, also called *development for reuse*, focuses on the development of core assets throughout the *domain analysis*, *domain design* and *domain implementation* processes. It is also responsible for defining the commonality and the variability of the product line. The analysis of the domain performed during this phase gives a set of requirements which can be reused to define the requirements of an application and to explicit the necessity to integrate new requirements. A reference architecture, defined by the domain design, is used to develop and structure applications. A backward and forward traceability must be established between the reference requirements, the reference architecture and the reusable components to facilitate the changes and updates management in the product line.

The *application engineering* phase, also called *development with reuse* or *product derivation*, consists of developing the final products using core assets and following specific customer requirements. It consists of three steps: *application requirements*, *application design* and

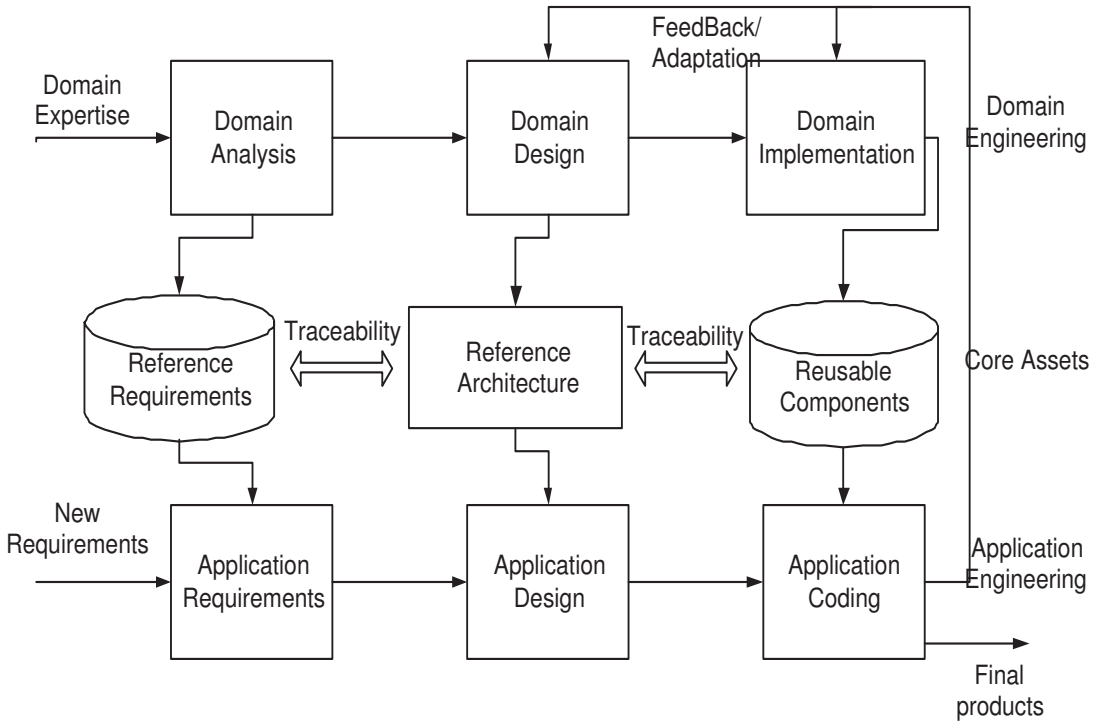


Fig. 2.1: General SPL engineering process [PBvdL05]

application coding. In this phase new systems are built based on the results of domain engineering. During this phase, a feedback process can be used to revise the domain design and the domain implementation. New products may reveal the necessity to integrate new reusable components to the product line's architecture or to modify reusable components.

There is a clear advantage of having a two phase process: a separation of the two concerns, to build a robust platform and to build customer-specific applications in a short time is achieved [PBvdL05]. In order to be effective, the two processes need to interact with each other in a manner that is beneficial to both. The two phases are actually intertwined: application engineering consumes the assets that are produced during domain engineering, while feedback from it facilitates the construction or improvement of the core assets. Figure 2.1 graphically represents the general SPL engineering process, as it can be found in the research literature [vdL02]. The two phases of the SPL engineering process are discussed in more detail in the following.

2.1.3 Domain Engineering

Domain engineering is the SPL engineering process phase in charge of *core assets development*. It follows a waterfall life cycle model. Its key goals, as satted by Pohl et al. [PBvdL05] are to:

- define the commonality and the variability of the software product line
- define the set of applications the SPL is planned for (define the scope of the SPL)
- define and construct reusable artefacts that accomplish the desired variability

Various inputs may be used for core assets development, like: production constraints and production strategy. Reuse is an important aspect of this phase. The goal is to reuse available pre-existing components, but also the development experiences of the company. The assets created during this phase describe partial solutions (such as a component or design document) or knowledge (such as a requirements database or test procedures) that engineers use to build or modify software products [Wit96].

Domain engineering also deals with *identifying the existing commonality and variability amongst SPL members.* Even if the SPL approach is a new paradigm, managing variability in software systems is not a new problem and some design and programming techniques allow to handle variability. However, outside of the SPL context, variability concerns a single product, and is resolved after the product is delivered to customers and loaded into the final execution environment. For software product lines, variability should explicitly be specified and be a part of the SPL. In contrast with single product variability, product line variability is resolved before the software product is delivered to customers. In [Nor99], the variability included in the single product is called *run time variability*, while product line variability is called *development time variability.* The topic of SPL variability is discussed in-depth in a separate sub-section.

2.1.4 Application Engineering

Clements et al.'s definition [CN01] of a software product line discussed previously also mentions that the set of software-intensive systems is developed from a common set of core assets in a prescribed way. This specific activity is known as *application engineering* or *product derivation* [ZJ06b].

The main goals of application engineering, as stated by Pohl et al. [PBvdL05], are to:

- achieve a high reuse of the domain assets when defining and developing a product line application
- exploit the commonality and the variability during the development of a product line application
- document the application artefacts and relate them to the domain artefacts
- bind the variability according to the application needs

According to the derivation technique used, currently available approaches to support product derivation can roughly be organized in two main categories: *configuration* and *transformation.*

- **Derivation by configuration:** product configuration or *software mass customization* [Kru06] originates from the idea that product derivation activities should be based on the parametrization of SPL core assets, rather than focusing on how individual products can be obtained. When all SPL members can be completely characterized, an automated derivation process can be devised. It relies on selecting product features according to the variants offered by the product line requirements description. Then, a configuration tool selects and assembles core assets automatically according to a decision model.

Several configuration-based approaches base their decision models on feature models [CHE05b, GFdA98]. In [KKL+98a] the FODA approach is extended in order to support the description of domain assets at the design level. This idea is also explored in [CHE05b] through the concept of *staged configuration*: every time the user makes a choice in the feature model, a new feature model is computed according to user choices at a lower stage. Product configuration based on feature modelling has also received commercial tool support: Pure::Variants [Pur] (provides a complete feature modelling environment integrated with IBM Eclipse IDE) and BigLever GEARS [GEA] (acts as a "bridge" between several product lines to configure a particular product).

There are also configuration approaches that do not base their decision model on feature models. In [BF06], a decision model is used to relate features to their realizing software and hardware assets and the contextual information which provides additional constraints in a single model. This decision model is part of the ConIPF [HWK+06] methodology.

Van Ommering et al. [vO02] designed an architecture description language called Koala to define the product architecture based on a pool of components that may be reused from different product lines. Product derivation is performed by assigning values for parameters and switches, then a compiler will automatically configure the product according to these values.

- **Derivation by transformation:** the introduction of Model Driven Engineering (MDE) techniques has played a major role in SPL engineering, especially for supporting product derivation, by providing *models* as useful abstractions to understand assets, and *transformations* able to use them as first-class artefacts for product generation.

  In [KMHC05] the authors propose to derive products by instantiating, via MDE transformation mechanisms, a framework embodying core assets on the basis of a decision model and according to the variants selected for a specific product.

  In [ZJ06b] Ziadi et al. emphasize product derivation at the design level, for both static and behavioural aspects. Static models are described in terms of UML class diagrams. The derivation process uses a decision model taking the form of a design pattern to display the variants available for each product. Behavioural derivation is based on the synthesis of state machines from scenarios.

  In [PKGJ08a], Perrouin et al. propose a product derivation process that is a trade-off between automation and flexibility. They demonstrate how, by combining well-known derivation approaches, it is possible to provide tool support automating a significant part of this process.

2.1.5 Benefits and disadvantages

In the following, we briefly outline the key factors that motivate the development of software under the SPL engineering paradigm:

- *Reduction of development costs:* this is achieved through the reuse of core assets in several different kinds of systems, which implies a cost reduction for each system.

- *Improved quality through reuse:* the core assets have to prove their proper functioning in different products. This implies extensive quality assurance, therefore a significantly higher chance of detecting faults and correcting them, thereby increasing the overall quality of all products.

- *Reduction of time to market:* for SPL the time to market is initially higher, as the common artefacts have to be created first. Nevertheless, after this initial phase, the time to market is considerably shortened, as many artefacts can be reused for each new product.

- *Reduction of maintenance effort:* whenever a core asset is modified, the changes can be propagated to all products in which the artefact is being used. This may be exploited to reduce maintenance effort.

- *Managing evolution and complexity:* thee are two close coupled aspects. The introduction of a new artefact into the platform, or the change of an existing one, gives the opportunity for the evolution of all kinds of derived products. The reuse of core assets throughout the product line reduces complexity significantly.

- *Benefits for the customers:* they obtain products adapted to their needs and wishes. Moreover, they can purchase these products at a reasonable price as SPL helps to reduce production costs. Additionally, customers get higher quality products.

Besides the clear advantages, the software product line approach has also its risks and disadvantages. The most important are discussed in the following:

- The introduction of a product line approach implies a major change in mentality and has a high impact in terms of time and costs. For these reasons, SPL does not appear accidentally and an explicit effort is required to initiate it.

- Changing an organisation's original mode of development from single product view to a product line approach entails a fundamental shift for the organisation and can bring resistance to changes.

- Another problem is the lack of software engineers that have a global view on the entire product line. There is a clear need for a software product line expert who knows perfectly the application domain, has enough responsibility, authority, experience and understanding of software product line theories.

- Capturing requirements for a group of systems may require sophisticated analysis and intense discussions to agree on the common requirements and the variation points to be defined

- SPL core assets must be designed to be robust and extensible so that they can be used across a range of product contexts. Often, components must be designed to be more general without loss of performance, or be made extensible to accommodate product variations.

2.1.6 Variability in SPL

Variability is seen as *"the key feature that distinguishes SPL engineering from other software development approaches"* [BFG+02]. In common language use, the term *"variability"* refers to *the ability or the tendency to change.* It is a central concern in SPL development [HP03] and covers the entire development life cycle, from requirements elicitation to product testing. Variability management is thus growingly seen as complex process that requires increased attention.

2.1.6.1 General notions

In a SPL context, the notion of *"variability"* has been defined in several ways.

- For Weiss et al. it is *"an assumption about how members of a family may differ from each other"* [DMW99].
- According to Bachmann et al. variability means *"the ability of a core asset to adapt to usages in different product contexts that are within the product line scope"* [BC05].
- For Pohl et al. it is the variability *"that is modelled to enable the development of customised applications by reusing predefined, adjustable artefacts"* [PBvdL05].

It is important also to clarify what the *goal of variability is.* It is of course desirable to enable fast and cost effective production of products, but to limit the goal to this does not suffice. Bachmann et al. assert that the overall goal of variability in a software product line is to *"maximize return on investment for building and maintaining products over a specified period of time or number of products"* [BC05].

With variability being an extensive research topic in SPL engineering, several possible classifications have been proposed.

- Halmans et al. [HP03] distinguish between *essential* and *technical* variability, especially at requirements level. Essential variability corresponds to the customer's viewpoint, defining what to implement, while technical variability relates to product family engineering, defining how to implement it.
- A classification based on the dimensions of variability is proposed by Pohl et al. [PBvdL05]: *variability in time* concerns the existence of different versions of an artefact, valid at different times; *variability in space* defines the existence of an artefact in different shapes at the same time.
- According to Pohl et al. [PBvdL05], variability is important to different stakeholders and thus has different levels of visibility: *external variability* is visible to the customers while *internal variability*, that of domain artefacts, is hidden from them.

The management of variability in a SPL is a *delicate and complex process.* Svahnberg et al. [SvGB05] have identified a minimally necessary set of steps to be taken to adequately complete this process:

- *identification of variability:* determine where variability is needed in the product line (list the features that may vary between products)
- *constraining variability:* provides just enough flexibility for current and future system needs
- *implementing variability:* selects a suitable variability realization technique based on the previously determined constraints
- *managing variability:* requires constant feature maintenance and re-population of variant features.

Several authors propose mechanisms to implement and manage variability especially at code level. Jacobson et al. [JGJ97] and Bachmann et al. [BC05] propose to use mechanisms like inheritance, extensions and extension points, parametrization, templates and macros, configuration and module interconnection languages, generation of derived components, compiler directives for this purpose. Recently, tagging approaches were also proposed, such as [BCH+10]. Svahnberg et al. [SvGB05] present a taxonomy of different ways to implement variation points, which they refer to as "variability realization techniques".

It is thus important to mention which are the problems that may affect the introduction, modelling and management of variability in a SPL. Bosch et al. [BFG+02] identify and discuss some of these issues:

- the need of a first-class representation for features and variation points, lacking from most variability modelling techniques, which makes it difficult to distinguish variability at the requirements and realisation level;
- dependencies between architectural elements and features are rarely made explicit;
- software configuration management tools fail to support important variability management aspects;
- lack of methods, techniques and guidelines to help selecting the optimal life cycle phase when variation points should be introduced, extended or bound;
- selection of variability mechanisms without considering their specific advantages and disadvantages.

For the past few years, several *variability modelling techniques* have been developed, which offer viable solutions to some of the above mentioned problems. Some of the most important proposals in product line variability modelling are discussed and evaluated in the following.

2.1.6.2 Modelling variability in SPL

As variability is extensively used in SPL engineering, variability related concepts can be gathered in a *separate, dedicated language*. In Model Driven Engineering (MDE), the concepts of a domain is explicitly captured in a *meta-model*. Working at the level of *models* and *meta-models* makes it possible to analyse and classify SPL variability modelling methods at a higher level of abstraction and objectiveness, and to extract general observations

valid for an entire class of approaches. Therefore, we propose a new classification framework that looks at variability modelling approaches from a high level of abstraction and provides a *model driven point of view*. We focus on identifying and analysing the central concepts used by a wide variety of variability modelling techniques and show how they relate to each other. The analysis is performed at two levels: *meta-model* and *model*.

SPLs are usually characterized by two distinct concepts: a *set of core assets* or reusable components used for the development of new products - the (*assets model*); a *means to represent the commonality and variability* between SPL members - the (*variability model*). Our classification is based on these two concepts.

We performed a thorough analysis of the research literature which indicated two major directions in SPL variability modelling:

- Methods that **use a single model to represent the SPL assets and the SPL variability**:
  - Annotate a base model by means of extensions: [Cla01, ZJ06b, GS08, dOJdSGHM05]
  - Combine a general, reusable variability meta-model with different domain meta-models: [MPL+09]
- Methods that **distinguish and keep separate the assets model from the variability model**:
  - Connect Feature Diagrams to model fragments: [PKGJ08b, CA05, LGB08, ATK09]
  - Orthogonal Variability Modelling: [PBvdL05, MPH+07]
  - ConIPF Variability Modeling Framework (COVAMOF): [SDNB04, SDH06]
  - Decision model-based approaches: [DGR10, MS03, SJ04, ABM00]
  - Combine a common variability language with different base languages: [HMPO+08]

In this proposed classification, the terms *assets meta-model* (AMM) and *assets model* (AM) cover a broad spectrum, which depends on the point of view of the different authors. Thus, these notions are further refined for each particular class of methods. Figure 2.2 summarizes the proposed classification and the newly introduced concepts. It briefly depicts what happens at both meta-model and model level for the identified classes of techniques. In the following, we present in more details this classification.

A. Single model to describe the SPL assets and the SPL variability:

This category contains techniques that *extend a language or a general purpose meta-model with specific concepts that allow engineers to describe variability*. The core characteristic of these approaches is the *mix of variability and product line assets concepts into a unique model*. Concepts regarding variability and those that describe the assets meta-model are combined into a new language, that may either have a new, mixed syntax, or one based on that of the base model extended by the syntax of the variability language. These properties apply at both meta-model and model level. We further distinguish two sub-categories:

A1. Annotate a base model by means of extensions [Cla01, ZJ06a, GS08, dOJdSGHM05]: standard languages are not designed to explicitly represent all types of variability. Therefore, SPL models are frequently expressed by *extending or annotating standard languages*

<table>
<tr><th>Technique Name</th><th colspan="2">Meta-model level</th><th colspan="2">Model level</th></tr>
<tr><td>1. Unique model (combined) for product line assets and PL variability</td><td colspan="2"></td><td colspan="2"></td></tr>
<tr><td>Annotating the base model by means of extensions</td><td colspan="2">AMM+V</td><td colspan="2">PLM
(conform to AMM+V)</td></tr>
<tr><td rowspan="2">Combine a general, reusable variability meta-model with base meta-models</td><td>AMM</td><td>VMM</td><td colspan="2" rowspan="2">PLM
(confirm to AMM+V)</td></tr>
<tr><td colspan="2">\ /
AMM+V</td></tr>
<tr><td>2. Separate (distinct) assets model and variability model</td><td colspan="2"></td><td colspan="2"></td></tr>
<tr><td>Connect Feature Diagrams to model fragments</td><td>AMM</td><td>VMM</td><td>AM</td><td>VM (FDM)</td></tr>
<tr><td>Orthogonal Variability Modelling (OVM)</td><td>AMM</td><td>VMM</td><td>AM</td><td>VM (OVM)</td></tr>
<tr><td>ConIPF Variability Modelling Framework (COVAMOF)</td><td>AMM</td><td>VMM (CVV)</td><td>AM</td><td>VM (CVV)</td></tr>
<tr><td>Decision model based approaches</td><td>AMM</td><td>VMM (DMM)</td><td>AM</td><td>VM(DM)</td></tr>
<tr><td>Combine a common variability language with different base modelling languages</td><td>AMM</td><td>VMM (CVL)</td><td>AM</td><td>VM (CVL)</td></tr>
</table>

Notation used:

AMM – assets meta-model
VMM – variability meta-model
AMM+V – assets meta model with variability
CVL – common variability language
DMM – decision meta-model
AM – assets model
VM – variability model
PLM – product line model
FDM – feature diagram model
DM - decision model

Fig. 2.2: Classification of SPL variability modelling approaches.

(models). The annotated models are the union of all specific models in a model family and contain all necessary variability concepts. Regarding our classification, we observe at meta-model level an assets meta-model enhanced with variability concepts (denoted *AMM+V*). The term assets meta-model (AMM) refers here to a base or domain meta-model. Then, at model level, product line models (denoted *PLM*) can be derived. They conform to the *AMM+V* defined at meta-model level. A typical example is the *extension of UML with profiles and stereotypes.*

A first approach from this category is that of Clauss [Cla01, CJ01]. He applies variability extensions to *UML Class Diagrams.* Clauss uses generic models in which he explicitly defines variability at particular points called *hot spots.* The extensions proposed are based on the notions of *variation points* (to locate variability) and *variants* (concrete way to realize that variability). The following stereotypes are used: ⟨⟨*variationPoint*⟩⟩ , ⟨⟨*variant*⟩⟩.

A second approach comes from Jézéquel et al. [ZJ06a, ZHJ03]. They define a set of *stereotypes*, *tagged values* and *structural constraints* and gather them in a *"UML profile for product lines"* [ZHJ03]. Initially, extensions for class diagrams are proposed: ⟨⟨*optional*⟩⟩ stereotype to denote *optionality*, using UML inheritance to model *variation points*, *constraints* that specify structural rules applicable to all models tagged with a specific stereotype. The profile is then extended for sequence diagrams.

We also mention the work of Gomaa et al. [GS08, Gom05] on *multiple-view product line modelling using UML.* The views proposed are: *use case model* (functional SPL requirements), *static model* (static structural SPL aspects), *collaboration model* (capture the sequence of messages passed between objects), *state chart model* (address dynamic SPL aspects). A multiple-view model is modified at specific locations, different for each view: *variation points* in the use case model, *abstract classes* and *hot spots* in the static model.

A2. Combine a general, reusable variability meta-model with different domain meta-models [MPL+09]: these approaches focus on the meta-model level, where a two-step process is applied. Initially, *two separate meta-models are created*: an assets meta-model and a general, reusable variability meta-model. They are then combined, resulting in a *unique assets meta-model extended with variability concepts.* The term AMM denotes here a domain meta-model. At model level, product line models can easily be derived.

A representative approach comes from Morin et al. [MPL+09]. They propose a *reusable variability meta-model describing variability concepts and their relations independently from any domain meta-model.* Using Aspect-Oriented Modelling (AOM) techniques, variability can be woven into any given base meta-model. A central concern of this method is the definition of a general variability meta-model, which is based on the work of Schobbens et al. [SHTB07].

B. Separate the assets model from the variability model:

Techniques in this category have *separate representations for the variability and for the assets model.* The key characteristic of such methods is the clear separation of concerns, which applies at both meta-model and model level. Elements from the variability model relate to asset model elements by referencing or other techniques. Advantages of these methods include: each asset model may have more than one variability model; designers can focus on the product line itself and not on its variability, addressed separately; possibility for a standardized variability model. We further identify three sub-categories of methods which share the same principle but differ in the type of variability model used.

B1. Connect Feature Diagrams to model fragments [PKGJ08a, CA05, LGB08, ATK09]:

despite their popularity, feature diagrams lack a lot of information. The need arises to combine them with other product representations. An emerging research direction is to associate model fragments to features. The FD defines the product line variability, with each feature having an associated implementation. Concerning our classification, there is a *clear distinction between assets and variability related concepts* at meta-model level. This situation extends at model level. For this category, the assets model is of a set of software artefact/asset fragments. The particular type variability model used is a *Feature Diagram (FD)*. A more detailed presentation of the techniques belonging to this category is available in one of the next sections.

B2. Orthogonal Variability Modelling [PBvdL05, MPH+07]: the assets model and the variability model are still kept separate. The variability model relates to different parts of the assets model using *artefact dependencies*. The differentiating factor is the type of variability model used: an *orthogonal variability model (OVM)*. There is also a difference regarding the assets model, which is now a compact software development artefact.

The OVM concept was introduced by Pohl et al. [PBvdL05] as *"a model that defines the variability of a SPL separately and then relates it to other development artefacts like use case, component and test models"*. It provides a view on variability across all development artefacts. The central concepts are *variation points (VP)*) and *variants (V)*. Both VPs and Vs can be either *optional* or *mandatory*. Optional variants of the same VP are grouped together by an *alternative choice*. The variability model relates to other software artefacts using *traceability links*. A special type of relationship called *artefact dependency*, relates a V or a VP to a development artefact.

B3. ConIPF Variability Modeling Framework (COVAMOF) [SDNB04, SDH06]: we include in this category the COVAMOF method of Sinnema et al. Concerning our classification, we identify separate variability and assets meta-models at the meta-model level. This reflects also at model level, where a separate variability model, called *COVAMOF Variability View (CVV)*, and an assets model can be distinguished.

The goal of COVAMOF is to uniformly model variability in all abstraction layers of a SPL. Variability is represented using *variation points* and *dependencies*. *Variation points* in the CVV reflect the variation points of the product family and are associated with product family artefacts. *Dependencies* are associated with one or more variation points and are used to restrict the selection of associated *variants*.

B4. Decision model based approaches: this class of approaches differs by using *decision models* as a particular type of variability model. For Bayer et al. a decision model *"captures variability in a product line in terms of open decisions and possible resolutions"* [BFG00]. Decision-oriented approaches treat *decisions* as first-class citizens for modelling variability.

A representative approach is DOPLER (Decision-Oriented Product Line Engineering for effective Reuse) from Dhungana et al. [DGR10]. It was designed to support the modelling of both *problem space variability* using decision models, and *solution space variability* using asset models and also to assure traceability between them.

There are other decision model based approaches except DOPLER. Schmid et al. [SJ04] extend the Synthesis approach with binding times, set-typed relations, selector types, mapping selector types to specific notations, using multiplicity to allow the selection of subsets of possible resolutions. The KobrA approach [ABM00] integrates product line engineering and component-based software design. KobrA decision models are described using a tabular notation.

B5. Combine a common variability language with different base languages [HMPO+08]: methods in this category propose a *generic language or model that subsumes variability related concepts.* The same general variability model can be combined with different base models, extending them with variability. Regarding our classification, at meta-model level there is a separate generic variability meta-model and an assets meta-model (AMM). At model level, variability model elements relate to assets model elements by referencing and using substitutions.

We mention the work of Haugen et al. [HMPO+08] who propose a simple domain specific language focusing only on variability, called *Common Variability Language* (CVL). CVL models specify both variabilities and their resolution. By executing a CVL model, a base SPL model is transformed into a specific product model. The CVL model points out base model elements and defines how they can be replaced to generate a new product model. The substitutions defined are: *value*, *reference* and *fragment.*

2.1.7 Feature Modelling

Central to the product line paradigm is the modelling and management of variability, the commonalities and differences in the applications in terms of requirements, architecture, components, and test artefacts [PBvdL05]. At all those levels, but especially at the requirement level, a popular way to model variability is through *Feature Models.* They are the first proposal of the SPL community for dealing with variability.

2.1.7.1 General concepts

In feature modelling, the notion of *feature* commonly refers to requirements, but can also denote domain properties, specifications and design, leading to confusion as to what exactly features describe. Several authors propose definitions for the notion of feature.

- Kang et al. give the most general one: *"a prominent or distinctive user-visible aspect, quality or characteristic of a software system or systems"* [KCH+90].
- Bosch specializes this definition for software systems: *"a logical unit of behaviour that is specified by a set of functional and quality requirements"* [Bos00]. From this point of view, a feature is a construct used to group related requirements.
- In this thesis we consider the definition of Czarnecki: *"a system property relevant to some stakeholder used to capture commonalities or discriminate among systems in a family"* [CE00].

In [SHTB07] Schobbens et al. distinguish between the specific use of feature models throughout the SPL engineering process:

- During domain engineering, features are *"units of evolution"* that adapt the system family to optional user requirements. A recurrent problem at this phase is the one of feature interaction: adding new features may modify the operation of already implemented ones;

- During application engineering, *"the product is defined by selecting a group of features, for which a carefully coordinated and complicated mixture of parts of different components are involved"* [Gri00]. It is therefore essential that features and their interactions are well-identified.

Feature diagrams emerged as a popular SPL variability modelling technique ever since Kang et al.'s [KCH+90] proposal in 1990 to express feature relations using a *feature model*. It consists of a *feature diagram (FD)* and other associated information: *constraints* and *dependency rules*. Feature diagrams provide a *graphical tree-like notation depicting the hierarchical organization of high level product functionalities* represented as features. The root of the tree refers to the complete system and it is progressively decomposed into more refined features (tree nodes). Relations between nodes (features) are materialised by *decomposition edges* and *textual constraints*.

Variability can be expressed in several ways. Presence or absence of a feature from a product is modelled using *mandatory* or *optional* features. Features can also be organised into *feature groups*. Boolean operators are used to select one, several or all the features from a feature group:

- *exclusive alternative(XOR)*: exactly one feature from a features group can be included
- *inclusive alternative(OR)*: one or more features from a set of features can be included
- *and*: all of the features will be included

Moreover, dependencies between features can be modelled using textual constraints. The most commonly use are:

- *require*: to express that the presence of a feature imposes the presence of another feature;
- *mutex*: to indicate that two features cannot be present simultaneously in the same product.

Feature diagrams are an essential means of communication between domain and application engineers, as well as customers and other stakeholders. They provide a concise and explicit way to:

- describe allowed variabilities between products of the same family;
- represent feature dependencies;
- guide the selection of features allowing the construction of a specific product;
- facilitate the reuse and the evolution of software components implementing these features.

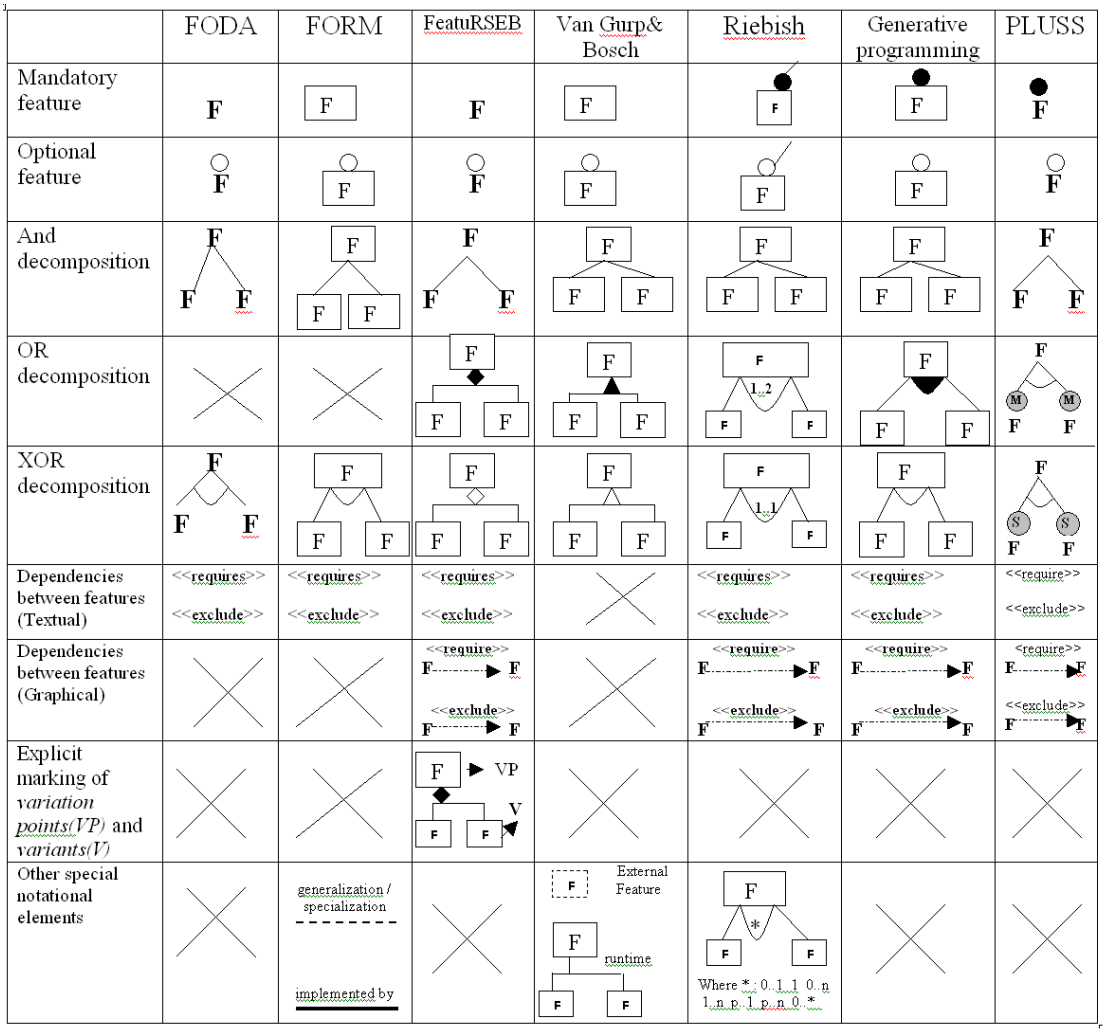


Fig. 2.3: Feature diagram dialects - synthesis of variability modelling concepts

2.1.7.2 Overview of feature modelling dialects

For the last 22 years, there have been a lot of contributions from research and industry in the area of feature modelling. The initial proposal of Kang et al. was part of the Feature Oriented Domain Analysis (FODA) methodology [KCH+90]. Its main purpose was to capture commonalities and variabilities at requirements level. This notation has the advantage of being clear and easy to understand. Unfortunately, it lacks the expressive power to model relations between variants or to explicitly represent variation points. Consequently, several extensions were added to this notation.

A first extension is the Feature Oriented Reuse Method (FORM) [KKL+98b] developed by Kang et al. in 1998. It proposes a four-layer decomposition structure, corresponding to different stakeholder viewpoints. There are small differences in the notation compared to FODA: feature names appear in boxes and three new types of feature relations introduced (*composed-of, generalization/specialization, implemented-by*).

Griss et al. propose FeatuRSEB [GFdA98], a combination of FODA and the Reuse-Driven Software Engineering Business(RSEB) method. The novelties proposed are: introduction

of UML-like notational constructs for creating FDs, explicit representation of variation points and variants (white and black diamonds), explicit graphical representation for feature constraints and dependencies. Van Gurp et al. [vGBS01] slightly extend FeatuRSEB by introducing *binding times* and *external features*.

Riebisch proposes the use of *UML multiplicities* [Rie03] in feature diagrams. *Group cardinalities* are introduced and denote the minimum and maximum number of features that can be selected from a feature group. There are two other changes: a feature is allowed to have multiple parents; edges are made optional or mandatory, not the features themselves.

Czarnecki et al. studied and adapted FD in the context of Generative Programming [CE00]. This proposal adds the *OR feature decomposition* and defines a *graphical representation of features dependencies*. More recently, the notation was extended with new concepts: *staged configuration* (used for product derivation) and *group and feature cardinalities* [CHE05a].

Finally, Product Line Use Case modelling for System and Software engineering (PLUSS) [EBB05] is an approach based on FeatuRSEB and combines feature diagrams and use cases. The novelty is changing the place of the decomposition operator, from the decomposed features or the edges, to the operand nodes. Two new types of nodes are introduced: *single adapters* (represent XOR-decomposition) and *multiple adapters* (OR decomposition).

In order to increase the clarity and conciseness of the methods previously presented, we provide in Figure 2.3 a synthesis of the concepts used to capture variability and how they are graphically represented by each feature modelling language. The figure shows what each feature modelling dialect is able to represent, as well as its limitations.

2.1.7.3 Associating models to features

Feature diagrams only provide a hierarchical structuring of high level product functionalities. One of the major downsides is that, using only feature diagrams, we are limited in the quantitative and qualitative information we can express. For example, there is no indication of what are the concrete representations of the features. Moreover, regarding product derivation, FDs only allow the SPL engineer to make a simple configuration of products through a feature selection. However, they tell very little about how the features are combined into an actual product. Due to these limitations, the need arises to combine feature models with other product representations. An emerging research direction is to *associate model fragments to features*. Different types of model fragment can be associated to features. The feature diagram defines the product line variability, with each feature having an associated implementation.

The first approach presented comes from Perrouin et al. [PKGJ08a], who address specific and unforeseen customer requirements in product derivation. The contribution of their work relevant to this thesis are two meta-models: a *generic feature meta-model* that supports a wide variety of existing FD dialects and a *subset of UML used to define the assets meta-model*. Based on the work of Schobbens et a. [SHTB07], Perrouin et al. extract a generic FD meta-model [Per06], with a simple and intuitive structure. Variability is represented using boolean operators. All classical feature diagram operators are provided: *or, and, xor, opt* and *card* to support group cardinalities. Feature dependencies like *mutex* or *require* can also be represented. In the feature diagram meta-model, the *Feature* meta-class is connected using a *composite association* to a class called *Model* that defines the core assets involved in feature realization. This relation specifies that a feature may

be implemented by several model fragments. Initially exploited with class diagrams, the meta-model allows any kind of assets to be associated with features.

Czarnecki et al. [CA05] propose a *general template-based approach for mapping feature models to concrete representations using structural or behavioural models*. The idea is to separate the representation of a product line model into: a *feature model* (defines feature hierarchies, constraints, possible configurations) and a *model template* (contains the union of model elements in all valid template instances). Elements of a model template can be *annotated*. These annotations are defined in terms of features from the feature model, and can be evaluated according to a particular feature configuration. Possible annotations are *presence conditions* (PCs) and *meta-expressions* (MEs). PCs are attached to a model element to indicate if it should be present or not in a template instance. Typical PCs are boolean formulas over a set of variables, each variable corresponding to a feature from the FD. MEs are used to compute attributes of model elements. When a PC is not explicitly assigned to an element of a model template, an implicit presence condition (IPC) is assumed. IPCs reduce the necessary annotation effort for the user. To derive an individual product (an instance of a model family), we must first specify a valid feature configuration. Based on it, the model template is instantiated automatically. To improve the effectiveness of template instantiation, the process can be specialized by introducing additional steps: *patch application* and *simplification*. The approach is general and works for any model whose meta-model is expressed in MOF.

There exist other methods belonging to this category, which we will only briefly mention. Laguna et al. [LGB08] separate SPL variability aspects using goal models and UML diagrams, while keeping features at the core of the representation. They combine previous approaches with the *UML package merge* implementation to provide a set of mapping rules from features to class diagram fragments. Apel et al. [ATK09] introduce *superimposition* as a technique to merge code fragments belonging to different features. They extend the approach and analyse whether UML class, state and sequence diagrams can be decomposed into features and then recomposed using superimposition to create complete models corresponding to SPL products.

2.2 Model Driven Engineering

Modelling is a cornerstone of all traditional engineering disciplines [Sel03]. From the conception and design, through the construction and maintenance of any engineered system, modelling plays a crucial role. Throughout the last decades software engineers have explored how lessons learned from traditional engineering disciplines can be applied to the design, construction, deployment and maintenance of software systems. The solution proposed is called Model Driven Engineering (MDE) [Ken02, Bez04, Fav04], a software development paradigm that raises the abstraction level for system specification and is highly regarded as a viable solution for building complex software systems.

Model-Driven Engineering is an approach to software development by which software is specified, designed, implemented and deployed through a series of *models*. The guiding principle of this new software engineering trend is to focus on models rather than on computer programs. According to Selic [Sel03], MDE is a good candidate to be the next established way to develop software: *"model-driven development holds promise of being the first true generational leap in software development since the introduction of the compiler"*.

From a historical point of view, MDE is a natural step in the evolution of software engineering, following the tendency towards raising the abstraction level in the design and development of software systems.

Model-driven initiatives propose a completely new terminology with a specific meaning.

- *Model-Driven Architecture (MDA): "is an OMG initiative that proposes to define a set of non-proprietary standards that will specify interoperable technologies with which to realize model-driven development with automated transformations"* [Gro03].
- *Model-Driven Development (MDD): "is simply the notion that we can construct a model of a system that we can then transform into the real thing"* [CJ03]. One difference with MDA is that MDD is not adhered to any of the OMG standards, according to Fowler [Fow09], but the main contribution of MDD is the flexibility offered to define development processes.
- *Model-Driven Engineering (MDE): "attempts to organize new efforts by proposing a framework (1) to clearly define methodologies, (2) to develop systems at any level of abstraction, and (3) to organize and automate the testing and validation activities".* [Ken02]. The MDE initiative proposes that any specification should be expressed by models, which are both human and machine understandable. Models, depending on what they represent, can reside at any level of abstraction, and can be restricted to address only certain aspects of the system.
- *Model-Driven Development (MDD): "is an emerging paradigm for software creation. It advocates the use of Domain Specific Languages (DSLs), encourages the use of automation, and exploits data exchange standards"* [Bat07].

Model-Driven Engineering promotes the systematic use of models as first class entities throughout the software engineering life cycle. The focus of development is shifted from third generation programming language codes to models expressed in proper domain specific modelling languages. The objective is to increase productivity and reduce the time to market by enabling the development of complex systems by means of models [Sei03].

Embracing this vision about MDE, the rest of this section focuses on: *model*, *meta-model*, *model transformation* and *model driven language engineering*, as they are crucial for an accurate understanding of MDE.

2.2.1 Models and meta-models

Models have become increasingly important in software engineering, and are a key concept and the main artefact in MDE [Sei03]. A model signifies a representation of some reality or system with an accepted level of abstraction, so all unnecessary details of the system are omitted for the sake of simplicity, formality, comprehensibility. A model has two key elements: concepts and relations. Concepts represent things and relations are the links between these things in reality. A model can be observed from different abstract point of views (*views* in MDE). The abstraction mechanism avoids dealing with details and eases re-usability.

The *model* concept is not a novelty. According to Favre [FEBF06], it dates back to ancient times, more than five thousand years ago. The word *"model"* has its etymological root

in the Latin word *"modullus"*, a diminutive of *"modus"*, which means a small measure. Today the interpretation of the word strongly depends on the point of view of the observer and his domain. The Merriam-Webster on-line dictionary gives 13 meanings of the word model. The first few of them are: *"a miniature representation of something"*; *"an example for imitation and emulation"*; *"a description or analogy used to help visualize something that cannot be directly observed"*.

If we restrain to the domain of software engineering, several authors have proposed definitions for the "model" concept. Some of them are presented in the following.

- A basic definition is given by Selic: *"a model is a representation of a system that hides some of the properties and highlights the ones that are of interest for the user". This hiding and highlighting means "a model is an abstraction"* [Sel06].
- Bezivin et al. define it as *"a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system"* [BG01].
- Kleppe et al. give a definition even more directed to MDE: *"a model is a description of a (part of) systems written in a well-defined language. A well-defined language is a language with well-defined form (syntax), and meaning (semantics), which is suitable for automated interpretation by a computer"* [KWB03].
- Seidewitz defines a model as *"a set of statements about a system under study. A model can be used either descriptively to determine properties of a system, or prescriptively as a specification of a system to be built"* [Sei03].
- The MDA guide defines a model of a system as *"a description or specification of that system and its environment for some certain purpose. A model is often presented as a combination of drawings and text. The text may be in a modelling language or in a natural language"* [OMG03].

Models thus provide simplified abstractions of the reality which encompass only the necessary details to the context taken into account. These abstractions allow domain experts to focus on the specific concepts related to their own domains leaving out non-essential aspects related to the chosen deployment platform. However, an abstraction is truly usefulness if it is complete and unambiguous for the purpose it has been conceived. The more models are precise, the easier it will be to produce useful artefacts and effective analyses.

In order to obtain the maximum benefits from the adoption of MDE techniques, it is necessary that all the needed information is represented by means of some kind of abstraction. This is leading to a new paradigm promoted by MDE, where "everything is a model" [Bez05]: requirements, tests, transformations and so forth are described as models. In MDE, we start from a description of a business feature by building models which are at high level of abstraction. The final goal is to get to the lowest level of abstraction, an executable system. In this manner, the understanding of both the business goals and the system under development evolves.

By modelling the target system, key problems can be revealed and the problem domain and its solution domain can be better described. Modelling allow us to record and describe the mapping of relationships from problem domains to solution domains and to resolve problems at the model stage [Sel03]. Resulting advantages include helping both developers

and users of the target system better understand, analyse and estimate system design accuracy and reliability and to discover the potential problems of system design by building a system model before the full system is created.

MDA defines three major classes of models, which refer to the development stages of software, going from the problem space to the implementation solution:

- *Computation Independent Model (CIM)*: is a view of the system from the computation independent viewpoint. According to the MDA guide, a CIM *"is a view of a system from the computation independent viewpoint. A CIM does not show details of the structure of systems. A CIM is sometimes called a domain model and a vocabulary that is familiar to the practitioners of the domain in question is used in its specification"* [OMG03].
- *Platform Independent Model (PIM):* is a view of a system from the platform independent viewpoint. A PIM *"exhibits a specified degree of platform independence so as to be suitable for use with a number of different platforms of similar type"* [OMG03].
- *Platform Specific Model (PSM):* is a view of a system from the platform specific viewpoint. A PSM *"combines the specifications in the PIM with the details that specify how that system uses a particular type of platform"* [OMG03].

One of the main motivations of this classification is to enable enterprises to preserve investments in business logic by means of a clear separation of the system functionalities from the specification of the implementation on a given technology platform.

Every model should conform to a *meta-model*. In the same way a grammar is specified to describe a programming language, a meta-model can be given to define correct models. Therefore, a meta-model can be considered as the set of rules to produce legal instances of a certain abstraction. The word *"meta"* is Greek and means *"above"*, therefore the term meta-model can be interpreted as a model describing another model.

When defining what a model is, Kleppe [KWB03] speaks about a *"well-defined language"* which can be used to create a model. In MDE meta-models define how a model can look like: a meta-model defines the constructs and rules usable to create a class of models. This is consistent with the following definitions:

- *"A meta-model is a model of a set of models"* [Fav05];
- *"A meta-model is a model that defines the language for expressing a model"* [Gro04].

From the above definitions we can deduce that a meta-model is a model itself, that is, a model of a language. Each meta-model defines the abstract syntax for a language by means of elements and relations between them. As such, a meta-model is, in turn, created using a modelling language. The meta-model used to define this meta-modelling language is referred to as the *meta-metamodel*. To avoid an infinite stacking of meta levels, meta-metamodels are often specified self reflexively and therefore the meta-model of the meta-metamodel is the meta-metamodel itself. This yields a layered architecture of models. In this respect, OMG has introduced the four level architecture which organizes artefacts in a hierarchy of model layers. This architecture is illustrated in Figure 2.4.

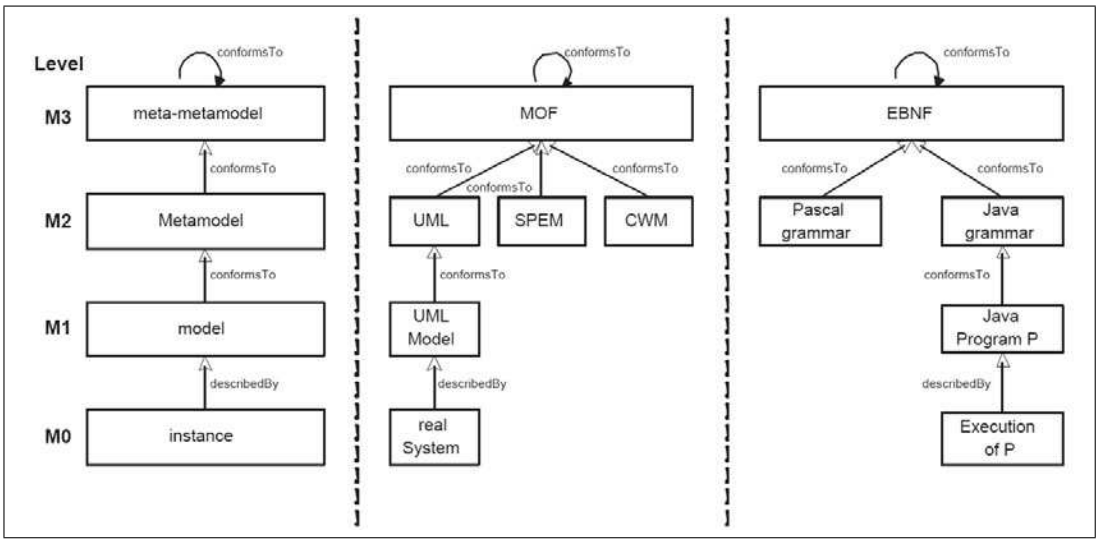


Fig. 2.4: The four layer meta-modeling architecture

- The M0 layer is an instance level. It is an example of the model in M1 level.
- The M1 layer is a model level. It is a model usually faced by the modelling people.
- The M2 layer is called the meta-model level and corresponds to the meta-model of the M1 layer. The M2 layer extracts abstract concepts and relative structure of different areas in the M2 meta-model. It also provides modelling symbols for the modelling language of the M1 layer. Therefore, the M2 layer provides corresponding domain-specific modelling language for different areas.
- The M3 layer holds a reflexively defined model of the information at M2, hence it does not require to refer to any further layers and is called the meta-metamodel. Eclipse Modeling Framework's (EMF) Ecore and Object Management Group's (OMG) Meta-Object Facility (MOF) are two well known meta-metamodels.

There are two important relations defined between elements from different layers of the previously described architecture. Elements from the M0 level are *"instances of"* or *"described by"* elements in the M1 level. Further, models from level M1 are *"conform to"* their meta-models from the M2 level. Similarly, all meta-models from M2 level are *"conform to"* the meta-metamodel from the M3 layer.

The importance and relevance of models and meta-models in MDE is further increased by *model transformations*. They are discussed in the following.

2.2.2 Model transformations

Working with multiple interrelated models requires significant time and effort to accomplish model management related tasks, such as refinement, consistency checking or refactoring. One of the major challenges related with the use of models in Software Engineering is to

automate these tasks. *Model transformations* have been accepted as the appropriate way to do so. Model transformations are essential for realizing the power of MDE [SK02, GLR+02].

The effectiveness of the MDE vision is fully attained through the use of model transformations. Transformations are the link between domain abstractions and represent a fundamental concern in development automation. A transformation is defined as a process that converts a source model into a target model related to the same system by means of a transformation specification [KWB03, OMG03]. In turn, a transformation specification encompasses the set of rules needed to map the source toward the target. Finally, each rule describes how to transform source instances to the corresponding target.

Model transformations are used for a variety of different purposes [CH06], including:

- generating lower-level models, and eventually code, from higher-level models;
- mapping and synchronizing among models at the same level or different levels of abstraction;
- model evolution tasks such as model re-factoring;
- reverse engineering of higher-level models from lower-level models or code.

The research literature offers several possible definitions for the *"model transformation"* concept:

- According to the OMG a model transformation is *"the process of converting one model to another model of the same system"* [OMG01];
- Kleppe et al. define model transformation as *"an automatic generation of the target model from a source model, which conforms to the transformation definition"* [KWB03];
- Tratt uses the following definition: *"a model transformation is a program which mutates one model into another; in other works, something akin to a compiler"* [Tra05].

In most of the above-mentioned definitions, a model transformation is regarded as a process that takes a model as input, referred to as the source model and produces as output another model, referred to as the target model. Figure 2.5 provides an overview of this process. The root of the process is the meta-metamodel (MMM). It provides with a set of basic abstractions that allow defining new meta-models. Next, the source and target meta-models are defined by instantiating the abstractions provided by the meta-metamodel. They are said to *conform to* the meta-metamodel. Finally, the *model transformation engine* executes the MMa2MMb *model transformation* to map an input model Ma into an output model Mb. To do so, MMa2MMb specifies a *set of rules* that encode the relationships between the elements from the MMa and MMb meta-models. The model transformation is defined at meta-model level, it maps elements from the input and output meta-models. Implicitly, it can be used to generate an output model from any set of models conforming to the input meta-model. In other words, the model transformation program works for any model defined according to the input meta-model.

If the set of rules and constraints that drives the construction of a model transformation are collected in a meta-model (MtMM), any model transformation can be expressed as a model

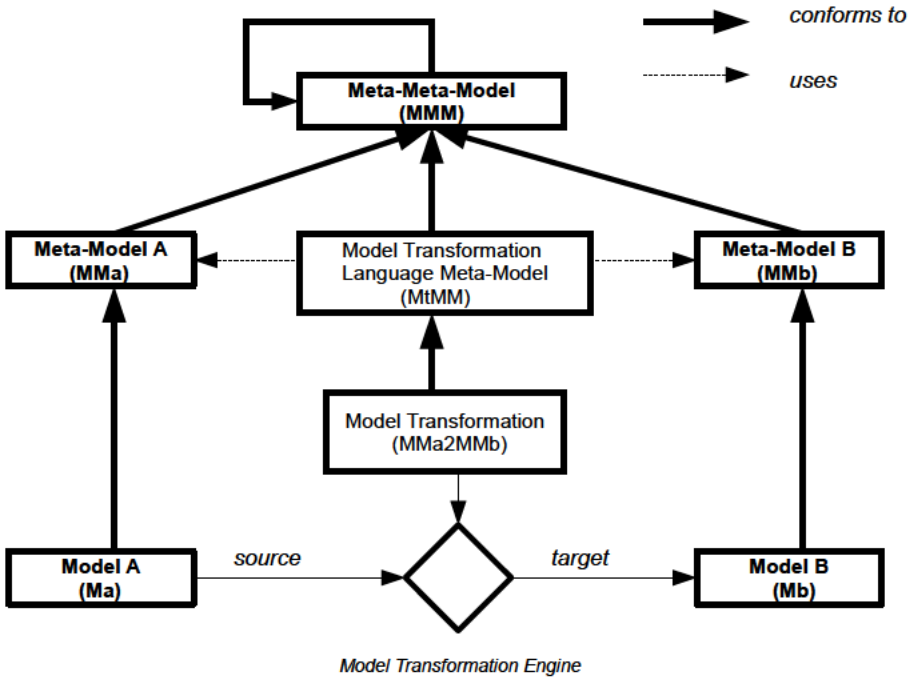


Fig. 2.5: Overview of Model Transformation process

conforming to such meta-model. Expressing model transformations as models, so-called transformation models, allows manipulating them by means of other transformations.

Model transformations are always implemented by an engine that executes the transformations based on a *set of rules*. The rules can be either *declarative* (outputs are obtained from some given inputs) or *imperative* (how to transform) [CH03]. Declarative rules are expressed in three parts: two *patterns* and a *rule body*. The two patterns are the source and target patterns respectively in a unidirectional transformation or the same pattern acting as source/target in a bidirectional transformation. A source pattern is composed of some necessary information about part of the source meta-model, according to which a segment of source model can be transformed. Similarly, a target pattern consists of some necessary information about part of the target meta-model, according to which a segment of target model can be generated. The link between these two patterns is the rule body. Declarative rules can be composed in a sequential or hierarchical manner, achieving flexibility and re-usability in transformations. Transformation have usually a mixed-style (having both declarative and imperative rules,) so that complex transformations can be implemented.

Transformations can be applied manually or automatically. In manual transformations, it is the developer's responsibility to investigate the input model and apply the modifications to it by adding, editing, or removing some model elements. Furthermore, the consistency of the resulting model is up to the developer. In automatic transformations, some transformation rules are defined to drive the changes, therefore the consistency of the output model is guaranteed.

MDE model transformations can be classified according to different point of views. Several

proposed classifications are available. In [MCVG05], a taxonomy of model transformations is discussed. Two orthogonal dimension pairs are defined: *horizontal versus vertical* and *rephrasing versus translation.*

- *Horizontal transformation* indicates transformation between different models at the same level of abstraction. Model refactoring is an example of such a transformation because the source model is restructured and the target models are at the same level of abstraction.
- *Vertical transformation* indicates a transformation where the source and target models reside at different levels of abstraction. Refinement is an example of such a transformation. The original model and its refined version are at different levels of abstraction.
- *Rephrasing* indicates a transformation where the models are expressed in the same modelling language. This kind of transformation is also called an endogenous transformation. Examples of rephrasing are optimisation, which aims at improving certain operational properties while preserving the semantics of the software, and re-factoring which aims at improving certain software quality characteristics while preserving the software's behaviour.
- *Translation* indicates a transformation where the source and target models are expressed in different languages. This kind of transformation is also called an exogenous transformation. Examples of translation are reverse engineering which extracts a higher-level specification from a lower-level one, and migration which translates a program written in one language to another, while keeping the same level of abstraction.

One of the main characterizations of model transformation approaches is the distinction between *model-to-text* (or *model-to-code*) and *model-to-model* techniques. A model-to-model transformation creates its target as a model which conforms to the target meta-model. On the contrary, the target of a model-to-text transformation is essentially strings.

- *Model-to-model transformations (M2M):* are one of the key aspects of MDE. In order for software systems to truly realize the potential of platform independence, models must exist for several target platforms and model to model transformations [BH02] must be designed to convert object models from one canonical form to any platform specific instance. MDE supporters expect that writing model to model transformations will become a common task in software development and these transformations will be shared among engineers [JK06]. As the name states, in model-to-model transformation, a model is changed to another model. The source model and the target model could be instances of the same meta-model or different meta-models. When both source and target are from the same meta-model, there are two specific cases of model-to-model transformations [BIJ06]: *refinement* and *refactoring.* In refinement transformations, a model is slightly changed to another model that better matches the desired system. Refinements can be done manually or automatically. In refactoring transformations, the designer tries to reorganize the model and make it simpler based on some well-defined criteria. In [CH06] a classification of model-to-model transformation languages is proposed. It is summarized in the following:

  - *Direct manipulation techniques:* users are provided with a minimum set of tools to implement transformation rules, scheduling, tracing and other facilities in a programming language. Usually, these techniques are based on an internal model representation and some APIs to manipulate it.
  - *Operational techniques:* a direct manipulation of input artefacts is allowed in some way similar to the previous category. In general, they are the result of an extension of the meta-modelling formalism which adds a set of features to express computations over the models. QVT Operational mappings [Gro05] or Kermeta [DFF+10] are examples of this methodology.
  - *Relational techniques:* declarative approach is based on the concept of relations. A type of relation between the source and target model must be first stated and then, it will be specified using constraints. Usually this specification is non-executable but it can check if two models are consistent. Therefore, relational approaches can be seen as a form of constraint solving methods. Examples of such approaches are QVT Relations [Gro05] and AMW [FBJ+05].
  - *Hybrid techniques:* different techniques from the previous categories are combined. For example ATL [JK06], which embodies imperative manipulations inside declarative constructs.
  - *Graph-transformation based techniques:* source and target models are represented as abstract syntax graphs on which transformation rules are based. In particular, each rule has a left-hand side (LHS) and a right-hand side (RHS). Both LHS and RHS describe graph patterns: when a LHS pattern is matched in the source model it is mapped toward the corresponding RHS pattern in the target. As a consequence, a rule without a RHS pattern deletes some source element and vice versa; that is, if a rule does not have an LHS pattern then some target element is created. AGG [Tae04], AToM3 [dLV02] and VIATRA2 [VVP02] are well-known graph-transformation approaches.

- *Model-to-Text transformation (M2T):* are also referred to as "code generation" or forward engineering techniques. Applying this transformation, part of the code is generated automatically from the model. Code generation is one of the features that distinguishes MDE from the old paradigms of software development. Most of the modern modelling tools are capable of generating code skeletons for a given model. The ultimate goal of MDE is to reach the level of full automatic code generation. There is evidence that this dream does not seem to be elusive, considering the advances in the supporting technology [Sel06]. There are two classes of M2T approaches:

  - *Visitor-based:* use the visitor design pattern to traverse across the whole model, translating each elements into code and printing it to a text stream.
  - *Template-based:* are more common in industry, than visitor-based ones. A template is usually a fragment of code with meta-code insertions to access a source model and get necessary information from it.

2.2.3 Model driven language engineering

In the context of software language engineering, Model Driven Engineering is beginning to take a more prominent role, due to the intensive use of models, considered as first

class artefacts of the development process, and of automatic model transformations, which drive the overall design, from requirements elicitation until the final implementation towards specific platforms. In this context, we talk about *model driven language engineering* [Ken03] when language development is carried out following the principles of the MDE approach: language descriptions are first class artefacts, and the abstract syntax of the language is defined in terms of a model, called language meta-model, which allows separating the abstract syntax and semantics of the language constructs from their different concrete notations.

Among software languages, a distinction can be made between *programming languages* [Tes84], used to develop software code running on a given platform and satisfying certain computational paradigms, and *modelling languages* [DH00], which are used for high level, platform-independent software design and are increasingly being defined as domain-specific languages (DSLs) [VDKV00] for specific domains of interest.

Traditionally, programming language construction follows a well-defined path [Tes84] usually consisting of the following steps:

- defining the language syntax: is mostly done using Bachus-Naur Form (BNF) [Nau63];
- generating a parser;
- defining a type-system;
- developing algorithms that walk the abstract syntax tree and check the well-typedness of the program.

In the model-based development context, meta-model-based languages are increasingly being defined and adopted either for general purposes or for specific domains of interest. Modelling languages offer designers modelling concepts and notations to capture structural and behavioural aspects of their applications. In contrast to general-purpose modelling languages (like UML) that are used for a wide range of domains, some modelling languages are often tailored to a particular problem domain, and for this reason considered domain-specific.

The development process of modelling languages diverges from the traditional language design [SK95], since modelling languages are usually introduced to model specific domain concepts, should be easy and fast to define, and should allow re-use of previously defined artefacts. Modelling languages themselves can be seen as artefacts of the model-based approach to software language engineering. In a model-based language definition, the *abstract syntax* of a language is defined in an abstract way by means of a meta-model, that characterizes syntax elements and their relationships, separating thus the *abstract syntax* and *semantics* of the language constructs from their different concrete notations. The definition of a language abstract syntax by a meta-model is well mastered and supported by many meta-modelling environments (Eclipse/Ecore [Ecl12], GME/MetaGME [GME11], AMMA/KM3 [ATL], XMF-Mosaic/Xcore [Xac], etc.).

Regardless of their general or domain specific nature, modelling languages share a common structure. They usually have a:

- *concrete syntax* (textual, graphical, or mixed);

- *abstract syntax*;
- *semantics* which can be implicitly or explicitly defined, and may be executable.

Formally, a modelling language L is defined [KSN05] as a five-tuple $L = \langle A, C, S, MC, MS \rangle$, consisting of the *abstract syntax A, concrete syntax C, semantic domain S, syntactic mapping MC and semantic mapping MS*. We detail in the following each part of this language definition.

The syntax can be divided into *abstract syntax* and *concrete syntax*. The abstract syntax describes the high-level structure of language elements and their relations. The concrete syntax defines the actual (textual or graphical) representation of the models, i.e. the language sentences.

The *abstract syntax* is defined by means of a meta-model representing in an abstract (and possibly visual) way concepts and constructs of the modelling language, and providing the means (usually constraints) to distinguish between valid and invalid models. The meta-model of a language describes the vocabulary of concepts provided by the language, the relationships existing among those concepts, and how they may be combined to create models. A meta-model based abstract syntax definition has the great advantage of being suitable to derive from the same meta-model (through mappings or projections) different alternative concrete notations (textual or graphical or both) for various scopes like graphical rendering, model interchange, standard encoding in programming languages, while still maintaining the same semantics. Therefore, a meta-model could be intended as a standard representation of the language notation.

A language can have one or more *concrete syntaxes*, textual or visual or mixed, derived from the meta-model, as notation to be used by language users to effectively write models conforming to the language meta-model. The concrete syntax must be treated with equal attention as the abstract syntax. It is crucial element of language design and deserves to be a separate element within the language description. If no agreement on concrete syntax would exist, anything could represent anything, and language users would no longer understand each other. The description of the concrete syntax and the description of the abstract syntax are separate entities belonging to one language description. Fondement et al. [FB05] use the formalism of meta-modelling for both. A separate meta-model representing concrete syntax elements is build and related to the abstract syntax meta-model via a model transformation. Xtext [Xte07] uses both meta-modelling and BNF. From an existing BNF grammar that represents the concrete syntax, a meta-model is generated that represents the abstract syntax. Languages often have multiple concrete syntaxes. There is a growing need for languages that have both a graphical and a textual syntax.

Furthermore, the language description should at least contain a *mapping from concrete to abstract syntax*, and preferably also from abstract to concrete syntax. The *syntactic mapping*, $MC : C \rightarrow A$, assigns syntactic constructs to elements in the abstract syntax. In the process of creating a language description either one can be chosen as starting point, the other being developed together with the mapping to the first.

The syntax of the language, specified by means of a meta-model, only defines the structure of the language. However, the semantic properties such as conditions over valid models and the behavioural semantics of a model are not specified. A semantics description is included in a language description because the language designer wants to communicate

the meaning of the language to other persons. Semantics descriptions of software languages are intended for human comprehension. The semantics can therefore be seen as the abstract logical space in which models, written in the given language, find their meaning. Semantics have an equally important role for a language definition as the structure of the language.

The language semantics is defined [HR04] by choosing a *semantic domain S* and defining a *semantic mapping* $MS : A \rightarrow S$ which relates syntactic concepts to those of the semantic domain. The semantic domain S and the mapping MS can be described in various ways, from natural language to rigorous mathematical specifications. Both S and MS should be defined in a precise, clear, and readable way. The semantic domain S is usually defined in some formal, mathematical framework. The semantic mapping MS is not so often given in a formal and precise way.

For the description of the language semantics, there are several possible existing approaches [Kle08, NN92, CCG09]:

- *Operational:* this approach directly manipulates the model. It therefore allows to stay in the same technical space and express the evolution of the model state in the same specific domain. It generally implies extending the initial meta-model with the informations that describes the state of model at execution. The meaning of each possible statement that can be written using the language's constructs is specified by rules (or axioms) that determine the induced computation of such statement when it is executed on a particular abstract machine. The abstract machine is characterised by a state, whereas the rules specify how the state is transformed by a statement written using the different language constructs.

- *Axiomatic:* requires to define a set of properties satisfied by the model in the different steps of its execution (like pre- and postconditions). It is usually not easy to fully specify the behaviour of the model in such a manner [9]. An axiomatic semantics can not be made automatically or easily executable. Using this approach, the meaning of each possible statement that can be written using the language's constructs is specified by giving rules of the form $\{P\}C\{Q\}$ that relate the state before (i.e. P) and after (i.e. Q) the execution of a statement (i.e. C).

- *Translational:* specified by translating (mapping) the current language into another language that is formally well defined and understood. It relies on a previously existing semantics defined on the target language. It implies translating constructs from the initial domain into the constructs of the formal target domain. One of the main reasons for which translational semantics are used is to take advantage of the facilities and tools available in the target domain (code generators, model-checkers, simulators, visualization tools, verification tools). To use translational semantics, the appropriate target domain has to be chosen, depending on the kind of property to be checked or tool to be used. This approach requires to define a meta-model for the target language, which may not already.

2.3 Business processes

Since the beginning of the industrial revolution, the main focus of the business and commercial worlds was on automating and improving production efficiency and reducing costs

[LDL03]. In the 1960s, the inefficiencies and inaccuracies of companies in terms of performance began to matter to the customers, so many of them had to improve their business to keep their customers. This moment triggered the awareness of organisations of the importance of business processes. It became clear that it was vital for their survival to let business processes be at the heart of the company. In other words, business processes became the key to a successful business. Even though it was in the 1960s when Levitt [Lev60] first mentioned the importance of business processes it was not until the 1990s that processes acquired a real importance in enterprise design. Authors such as Harrington (1991), Davenport (1993) and Hammer (1990), among others, promoted the new perspective.

Experts in the fields of Information Technology and Business Engineering have concluded that successful systems start with an understanding of the business processes of an organisation. Furthermore, business processes are a key factor when integrating an enterprise [ASO05]. A business process oriented perspective allows software architects and organizations to identify and reason about actors, goals, cooperation, commitments and customer relations, aspects which are crucial in a world of constant change for keeping the organizational objectives, and the objectives of the supporting information system aligned [MWH99].

A business process begins with a mission objective and ends with the achievement of the business objective. Business Processes are designed to add value for the customer and should not include unnecessary activities. The outcome of a well designed business process is increased effectiveness (value for the customer) and increased efficiency (less costs for the company). Business processes use information to tailor or complete their activities. Information, unlike resources, is not consumed in the process - rather it is used as part of the transformation process. In formation may come from external sources, from customers, from internal organisational units and may even be the product of other processes.

There are several possible definitions for the notion of *"business process"* available in the research literature:

- Davenport defines a business process as *"a structured, measured set of activities designed to produce a specific output for a particular customer or market. It implies a strong emphasis on how work is done within an organization. A process is thus a specific ordering of work activities across time and space, with a beginning and an end, and clearly defined inputs and outputs. Taking a process approach implies adopting the customer's point of view. Processes are the structure by which an organization does what is necessary to produce value for its customers"* [DS90];

- Hammer et al. define it as *"a collection of activities that takes one or more kinds of input and creates an output that is of value to the customer"* [HC03]. This definition provides a more transformation oriented perception, and puts less emphasis on the structural component;

- According to Scheer et al. *"the term business process is intended to embrace not only the control flow, i.e. the chronological sequence of function execution, but also the descriptions of data, organizations and resources that are directly associated with it"* [SAJK02];

- According to Weske *"a business process consists of a set of activities that are performed in coordination in an organizational and technical environment. These activities jointly realize a business goal. Each business process is enacted by a single*

organization, but it may interact with business processes performed by other organizations" [WGHS99];

- In the context of this thesis, we retain the definition provided by the Workflow Management Coalition (WfMC) in its "Terminology and Glossary": *"a business process is considered as a set of one or more linked procedures or activities which collectively realise a business objective or policy goal, normally within the context of an organisational structure defining functional roles and relationships"* [Coa99].

There are three types of business processes:

- *management processes:* govern the operation of a system
- *operational processes:* constitute the core business and create the primary value stream
- *supporting processes:* support the core processes

As we have seen, business processes are the key to a successful business. This is probably the case as they focus on creating value for customers. They alone are not the solution, though. We need also Business Process Management (BPM) to contribute to this concept.

2.3.1 Business process management

Business Process Management (BPM) is an established discipline for building, maintaining, and evolving large enterprise systems on the basis of business process models [BKR03]. Organizations attempt to improve their business performance by applying BPM methods. BPM has become an essential way of controlling and governing business processes. For organizations it is generally important to discover, control, and improve their processes to increase their total revenue, their customer satisfaction or to ensure regulatory compliance as in the introductory example. BPM offers viable solutions to these aspects and deals with the coordination of activities in business processes within and between organizations.

The foundation of business process management lies in business administration and information systems. Business process management solutions have emerged in both industry products and academic prototypes since the late 1990s, when new innovations and technologies paved the way for BPM and its automation. Workflow management [JB96] was invented and new ideas brought into the BPM area. Methods like business process re-engineering [GKT93] and business process improvement [Har91] were adopted by commercial vendors and helped to analyse and optimize existing business processes. The organizations recognized that their business processes became more efficient and consequently started using these systems.

Nowadays, business process management is considered to support many aspects concerning business processes in and among organizations. These aspects include, e.g, advanced reporting and analysis technologies, quality assurance of processes, the automatic execution of processes with workflow management or the optimization and redesign of business processes. BPM allows organizations to abstract business processes from technology innovations and enables them to change their own business quickly according to their changed needs, customers, or regulatory compliance.

Many scientist and associations have created and proposed their own definition of business process management. A selection of the most relevant ones is presented here:

- Weske states that business process management *"includes concepts, methods, and techniques to support the design, administration, configuration, enactment and analysis of business processes"* [Wes07]. Thus, BPM can be seen as a holistic managing approach for the handling of business processes;
- In [vdAtHW03] BPM is defined as a management discipline *"supporting business processes using methods, techniques, and software to design, enact, control, and analyse operational processes involving humans, organizations, applications, documents and other sources of information"* [Wes07];
- Zairi suggests that BPM is *"the way in which key business activities are managed and continuously improved to assure consistent ability to deliver high quality standards of products and services"* [Zai97];
- Miers states that BPM *"should be thought first and foremost as a management philosophy that is driven from the top of the organization. It is not a new technology, rather it is a way of thinking that regulates the structure of the business and drives its overall performance"* [Mie09].

Business Process Management manages the life cycle of processes with respect to improvement and optimization to strengthen the ability to achieve the company's goals in an environment with growing complexity [SF03]. These management activities can be can be arranged in a life cycle, which consists of phases that are related to each other based on their logical dependency. All BPM activities can be attributed to one of the phases of the BPM lifecycle [ZM04]:

- *Analysis:* the BPM lifecycle begins with the analysis of a certain situation. During the analysis the organization and the process structure is investigated to conclude and derive requirements
- *Design:* the requirements serve as important input for the design phase. Within this phase, the business processes are identified. The process characteristics, the resources, the order of activities and organizational aspects are determined. The details are usually documented in the *modelling process* with the help of *business process models*. The models serve as representation of the real world processes.
- *Implementation:* the process models serve as input for the implementation phase. Based on the information in the models, the infrastructure for the business processes is set up. For the automatic execution of business processes, the process model serves as blueprint for the configuration.
- *Enactment:* in this phase the business processes run on the technical infrastructure as set up in the previous phase. Individual cases are handled by the system. Information about all cases, e.g. time data or resource allocation, is stored in the infrastructure. This data serves as input for the monitoring and evaluation phase.
- *Monitoring:* the monitoring of the processes in the system is important for identifying deviations or problems at an early stage. The monitoring can be done automatically. The infrastructure can also execute countermeasures to fix certain problems or deviations once they are detected by the system.

- *Evaluation:* this phase compares the actual process data handled by the systems with the requirements stated in the analysis phase. In the evaluation, new requirements might come up which are then fed back to the design phase to change the business processes and the corresponding process models. Thus, the overall performance of the business processes is measured and continuously improved.

From the phases in the BPM lifecycle, it becomes obvious that business process models play an important role in the lifecycle.

2.3.2 Business process modelling

Business process modelling is a key phase of the BPM lifecycle, which intends to separate process logic from application logic, such that the underlying business process can be automated [BSW04]. The modelling of business processes is becoming increasingly popular and plays a pivotal role in the business process management discipline. Both experts in the field of Information and Communication Technology (ICT) and of Business Engineering have concluded that successful systems (re)engineering starts with a thorough understanding of the business processes of an organisation: a *business process model.*

Business process modelling is a widely-used approach to achieve the required visibility for existing processes and future process scenarios. It claims a more disciplined, standardized, consistent and overall more mature and scientific approach. It facilitates process visibility and has to satisfy an increasingly heterogeneous group of stakeholders and modelling purposes. It has to be scalable, configurable and usually able to provide a bridge between IT capabilities and business requirements. It is an essential part of any software development process, as it allows the analyst to capture the broad outline and procedures specifying what a business does. This model provides an overview of where the proposed software system being considered will fit into the organisational structure and daily activities. As an early model of business activity, it allows the analyst to capture the significant events, inputs, resources and outputs associated with business process.

Real-world or artificial business processes are mapped into business process models. This explicit representation is an essential concept within business process modelling. It helps achieve the communication among stakeholders and creates a common understanding of the processes [Wes07]. Business process modelling is therefore the human activity of creating business process models. They have become an integral part of the organizational engineering efforts. A business process model is simply a flow-oriented representation of a set of work practices aimed at achieving a goal. More formally, Mendling [Men07] defines a business process model as "the result of mapping a business process. This business process can be either a real-world business process as perceived by a modeller, or a business process conceptualized by a modeller".

Business process models are used on the business level for describing business operations in a consistent way, as well as on the technical level for specifying requirements that have to be supported by enterprise software. In practice, business process models are often used for documentation purposes and business process design is one of the major reasons for conducting conceptual modelling projects [DGR+06]. Therefore, most business process models can be regarded as descriptive models for organization, although they may also serve as decision models in other phases of the BPM life cycle.

Business process models show an abstract view of complex structures, which has a number of advantages [CKO92]:

- meaning of each process is precisely defined
- models are graphical and therefore easy to understand, which allows different users to interpret them in the same way
- new processes can be modelled by combining existing processes or components in new ways
- allow to focus on a specific part of a structure, in such a way that key relationships are highlighted and less relevant aspects ignored

The increasing popularity of business process modelling resulted in a rapid growing number of modelling techniques. Several languages have been proposed for business process modelling. Though most of them follow the conventional representation of processes as a series of steps, they emphasize different aspects of processes and related structures, such as organizations, products, and data. We provide in the following a brief overview of the languages that have played an important role in business process modelling.

Flowchart: a flowchart is defined as *"a formalised graphic representation of a program logic sequence, work or manufacturing process, organisation chart, or similar formalised structure"* [LCB96]. It is a diagram that represents a process as a sequence of activities and decisions. Flowcharts are the oldest and most basic process related modelling methodology known, with their first reported occurrence dating back to the early 1920s, where they were used by mechanical engineers to describe machine behaviour. Basic flowchart constructs are activities, decisions, start points and end points. These are the basic building blocks typically used to represent processes. More advanced flowcharts use data-flow constructs which denote the information that flows throughout the process. Relationships in a flowchart are denoted by arrows which indicate a flow of control from one element to another. All elements in a flowchart are either directly or indirectly connected with one another.

Role activity diagrams (RAD): are based around a graphic view of the process from the perspective of individual roles, concentrating on the responsibility of roles and the interactions between them [HRG83]. The primary constructs used in a RAD are roles, actions, interactions and decisions. Roles contain the actions and decisions that are performed by the man or machine with the assigned role. The interaction construct allows a role to communicate with another role, which also constitutes the only way how a relationship can be established between roles.

UML Activity Diagrams (UML AD): are part of the Unified Modeling Language (UML) [Gro07]. UML was primarily designed to model software systems, however some diagram types, such as UML ADs, can also be used for business process modelling. A business process can be described by an activity consisting of a coordinated sequencing of nodes, based on control-flow and object-flow. The control-flow comprises two types of nodes: action nodes and control nodes. An action node can model an activity to be performed or a signal to be received/sent by the process. Control nodes are used to model sequencing and parallel or alternative branching. Processes can be organized in a hierarchy by means of compound activities, in order to avoid cluttering the model. There are two other important features of UML AD which are swimlanes and sub-activities. Swimlanes

can be used to group actions on some common characteristic. Sub-activities can be used to aggregate an activity diagram into a single activity for use in other activity diagrams. Sub-activities facilitate composition and decomposition in activity diagrams.

Event-driven Process Chains (EPC): are an easy-to-understand language for modelling business processes [SL05], initially developed for the design of the SAP R/3 reference process model [Her97]. EPCs also became the core modelling language in the ARIS platform [Sch00]. Since its creation, the EPC method has grown to become one of the more popular business process modelling methodologies. An EPC is a directed graph consisting of events, functions, connectors and arcs linking these elements. Each EPC starts and ends with at least one event. Events are triggers for functions and signal their completion, while functions represent activities to be performed. Each function is preceded and followed by an event. Connectors are used to model alternative and parallel branching and merging. They are splits and joins of the logical types of OR and XOR (for inclusive and exclusive decision and merging, respectively) and AND (for parallelism and synchronization).

Web Services Business Process Execution Language (WSBPEL): web services play an important part in the BPM landscape. BPEL [OAS07] is used to describe the behaviour of Web services using business process modelling constructs. For this reason, BPEL represents a convergence between Web services and business process technology. BPEL extends imperative programming languages, like C, with constructs for the implementation of Web Services. A BPEL process is exposed as a Web Service through WSDL interfaces. A BPEL process is a hierarchical structure of basic activities corresponding to atomic actions for sending, receiving and creating/processing messages. Compound activities determine the process structure by allowing sequential, parallel and conditional routing, as well as looping. It is also possible to specify events as external agents, such as a time-out or a message receipt. Specific activities are also available for exception handling and recovery. The language has been designed to specify both abstract and executable processes.

Yet Another Workflow Language (YAWL) [vdAtH05]: is an expressive language to describe, analyse and automate complex business process specifications, built on top of the research outcomes of the Workflow Patterns Initiative [Ini11]. YAWL was realized by extending Petri nets with vital constructs to directly support the workflow patterns. Nevertheless, YAWL is a completely new language with a formal semantics specifically designed to model workflow specifications. A YAWL model is a hierarchical structure of tasks corresponding to atomic or composite work items (similar to transitions in Petri nets), and conditions, to explicitly represent the notion of state. Splits and joins are of type OR, XOR and AND, and are defined as output, respectively, input decorations of a task. Multiple instance tasks and cancellation regions complete the control-flow semantics of YAWL, and are used to model advanced control-flow features. YAWL relies on global variables to capture the data-flow.

Other important process modelling languages like *Petri Nets (PN)* and the *Business Process Modelling Notation (BPMN)* will be addressed in more detail in the following sections.

2.3.3 Business Process Modeling Notation

The Business Process Modeling Notation (BPMN) [OMG11] is gaining adoption as a standard notation for capturing business processes [RIRG05]. The initial version of BPMN was developed by the Business Process Management Initiative (BPMI) in 2004. Two years

later, the Object Modeling Group (OMG) adopted the language as a standard for business process modelling [Gro06a]. As of 2011, BPMN is the most used notation for the modelling of business processes and considered the de facto standard [18]. Currently, the latest version of BPMN is 2.0 [OMG11].

The primary goal of BPMN is to provide a notation that is easily understandable by all business users, starting with the business analysts that create the initial drafts of the processes, to the technical developers responsible for implementing the technology that will perform those processes, and finally, to the business people who will manage and monitor those processes. BPMN aims to support the complete range of abstraction levels, including business levels and software technology levels. Thus, BPMN creates a standardized bridge for the gap between the business process design and process implementation and also between technical and non-technical people. In spite of being easy to use, BPMN also has the ability to model very complex processes.

A second, equally important goal of the language, is to ensure that XML-based languages designed for the execution of business processes, such as BPEL4WS and BPML, can be visually expressed using a common notation. It also tries to be formal enough to be easily translated into executable code. By being adequately formally defined, it can create a connection between the design and the implementation of business processes. BPMN is based on the same principles as flowcharts, but includes a much greater variety of constructs, making the language far more expressive than flowcharts. Besides flowcharts, the constructs present in BPMN have their roots and are inspired from other notations and methodologies, especially UML Activity Diagram, UML EDOC Business Process, IDEF, ebXML BPSS, Activity-Decision Flow (ADF) Diagram, RosettaNet, LOVeM and EPCs.

The main concept specified in BPMN is a single diagram, called the Business Process Diagram (BPD) [WS05], which can be used to create graphical models especially useful for modelling business processes and their operations. It is based on a flowchart technique - models are networks of graphical objects (activities) with flow controls between them. The purpose of this diagram is twofold. First, it can quickly and easily be used to model business processes, and it is also easily understandable by non-technical users (usually management). Second, it offers the expressiveness to model very complex business processes, and can be naturally mapped to business execution languages.

Business Process Diagrams were developed with web services and the Business Process Execution Languages (BPEL) in mind. Thus, they map directly to any major execution language such as: Business Process Execution Language for Web Services (BPEL4WS) or Business Process Modeling Language (BPML). We will describe these execution languages in more detail later. BPD also serve as a common visual notation for expressing different execution languages.

A BPD is made up of a set of graphical elements [Whi04]. These elements facilitate the development of simple diagrams that will look familiar to most business analysts (e.g., a flowchart diagram). The elements were chosen to be distinguishable from each other and to utilize shapes that are familiar to most modellers. For example, activities are rectangles and decisions are diamonds. It should be emphasized that one of the drivers for the development of BPMN is to create a simple mechanism for creating business process models, while at the same time being able to handle the complexity inherent to business processes. The approach taken to handle these two conflicting requirements was to organize the graphical aspects of the notation into specific categories. This provides a small set of notation categories so that the reader of a BPD can easily recognize the basic elements and

understand the diagram. Within the basic categories of elements, additional variation and information can be added to support the requirements for complexity without drastically changing the basic look-and-feel of the diagram. The four basic categories of elements are:

- *Flow Objects*;
- *Connecting Objects*;
- *Swimlanes*;
- *Artifacts*.

Flow Objects are the core of a BPD and contain three types of objects defining the behaviour of a business process:

- *Events:* an event is something that happens during the course of a business process. These events affect the flow of the process and usually have a cause (trigger) or an impact (result). Events are graphically represented by circles with open centres to allow internal markers to differentiate different triggers or results. The event notion covers a broad spectrum of concepts in a business process fragment, like: the start or end of an activity, the sending of reception of a message, the occurrence of an error, the end of a time interval. There are three main types of events, based on when they affect the flow: *start*, *intermediate* and *end*.
  - *Start:* indicates where a particular business process starts. In terms of connection with other flow objects, the start event initiates the flow of the business process, and therefore will not have any incoming sequence flow connections. It is mandatory for every business process model to have a unique start event. Implicitly, this is the only entry point to the business process;
  - *Intermediate:* denotes that something happens inside the flow of the business process. Intermediate events will affect the flow of the business process, but will not start or terminate it. This type of events are generally used for modelling message exchanges, delays are expected within the process or the occurrence of errors during the flow of a business process;
  - *End:* indicates where a business process will finish. In terms of connection with other flow objects, the end event terminates the flow of the business process, and therefore will not have any outgoing sequence flow connections. End events are mandatory within a business process.

  Every event has a *trigger*, which defines the cause for that event. There are multiple ways in which an event can be triggered. For a start event, the triggers are designed to show the general mechanism that will instantiate that process fragment. They may also define the consequences of reaching an end event. The BPMN standard proposes a set of 10 types of event triggers. The most important ones are:
  - *Message:* can be applied to any type of event. For the start event, it denotes the arrival of a message from a participant and triggers the start of the business process. If applied to an end event, it indicates that a message is sent to a participant at the conclusion of the business process. Finally, when applied to

an intermediate event, it indicates that a message arrives from a participant and triggers the event. This causes the business process to continue if it was waiting for the message, or changes the flow for exception handling.

- *Timer:* can only be applied to start or intermediate events. It may denote that a specific time-date or a specific cycle can be set that will trigger the start of the business process. If used within the main flow, it acts as a delay mechanism.
- *Plain:* it is the most generic type of trigger and can be applied to any type of event. The modeller does not display the exact cause of the event. Within the main sequence flow, it is used to indicate a change of state in the business process;
- *Error:* this type of trigger can be assigned to intermediate and end events. They signal an error in the functioning of the business process and disrupt the normal flow of activities. Error events can be addressed in several ways, depending on the type of error handling mechanism applied.

- *Activities:* an activity is a generic term for work that company performs. Activities are the main elements of a business process. It is represented by a rounded-corner rectangle. We distinguish between *atomic* and *compound* activities. The types of activities that are a part of a business process model are: *task* and *sub-process*.

  - *Task:* is an atomic activity that is included in a business process. They are mostly used when the behaviour described by the business process is not broken down to a finer level of detail. In general, an end-user and/or an application are used to perform the task when it is executed.
  - *Sub-process:* is a compound (non-atomic) type of activity. It has detail that is defined as a flow of other activities. Sub-processes are complex activities which require several atomic activities to be performed/executed. Implicitly, a sub-process consists of several tasks. A sub-process is characterized by its type, which can be either *collapsed* or *expanded*. A collapsed sub-process hides its internal details. It therefore only provides a high-level view of an activity, without detailing its internal mechanism. They are also used for providing a hierarchical organization of activities in a process fragment. On the contrary, an expanded sub-process shows its details within the view of the process fragment in which it is contained. They can be used to flatten a hierarchical process fragment so that all detail can be shown at the same time. They can also be used to create a context for exception handling that applies to a group of activities.

- *Gateways:* are used to control the divergence and convergence of sequence flow. Thus, they determine the branching, forking, merging, and joining of paths in a business process. They are used to control how the sequence flows interact as they converge and diverge within a process fragment. Gateways are not required if the flow does not need to be controlled. There is a mechanism that either allows or disallows the passage through every gateway. Thus, sequence flows that arrive at a gateway can be merged together on input and/or split apart on output as the mechanisms are invoked. Four different types of gateways exist. The behaviour of each type of gateway will determine how the sequence flow will continue after passing the gateway. A particular type of gateway can have multiple input and multiple output sequence flows at the same time. The type of gateway will determine the same type of behaviour for

both the diverging and converging sequence flow. There are four possible types of gateways:

- *Exclusive forking:* can be both *data-based* and *event-based*. They define locations within a business process where the sequence flow can take two or more alternative paths. This basically creates a forking of paths for a process fragment. However, only one of the paths can be taken. The choice of which path to follow is made based on a *decision*. A decision can be thought of as a question that is asked at that point in the process. The question has a defined set of alternative answers, each associated with a condition expression found within an outgoing sequence flow. When a particular alternative is chosen during the performance of the process fragment, the corresponding sequence flow is then chosen;
- *Inclusive forking:* represents a branching point where alternatives sequence flows may be followed. However, in this case, the evaluation to True of one condition expression does not exclude the evaluation of other condition expressions. All sequence flows with a True evaluation will be traversed. Since each path is independent, all combinations of the paths may be taken, from zero to all. However, it should be designed so that at least one path is taken;
- *Complex:* are used to handle situations that are not easily handled through the other types of gateways. They can also be used to combine a set of linked simple gateways into a single, more compact solution.
- *Parallel forking:* to define the parallel nature of this gateway's behaviour for splitting, if there are multiple outgoing sequence flow, all of them will be used to continue the flow of the business process. For the merging behaviour, all the incoming sequence flows will be synchronized;

Connecting Objects are used for connecting together the flow objects in a diagram, to create the basic skeletal structure of a business process fragment. They this define how the flow progresses through a process fragment (in a straight sequence or through the creation of parallel or alternative paths). There are three possible types of connecting objects:

- *Sequence Flow:* is used to show the order (sequence) in which flow objects will be performed in a business process. It is represented by a solid line with a solid arrowhead. Sequence flow will generally flow in a single direction (either left to right, or top to bottom). A sequence flow has only one source and only one target.
- *Message Flow:* is used to show the flow of messages between two separate business process participants (business entities or business roles) that send and receive them. In BPMN two separate pools in the diagram will represent the two entities between which the message exchange is performed. It is mandatory that message flow connects two flow objects that belong to different pools. The source and the target cannot connect two objects within the same pool. In case there is an expanded sub-process in one of the pools, then the message flow can be connected either to its boundary or to one of the flow objects within the sub-process.
- *Association:*: an association is used to associate extra information with Flow Objects. Text and graphical non-flow objects can be associated with flow objects. An association is represented by a dashed line with an open arrowhead.

BPMN specifies two ways of grouping modelling elements (e.g. progresses, events and gateways) through so called *Swimlanes*. They are used to help partition and/organize the activities of a business process. Their goal is to represent participants of a business process and their collaboration. Swimlanes may be arranged horizontally or vertically. They are semantically the same, just different in representation. For horizontal swimlanes, process flows from left to right, while vertical swimlanes flow from top to bottom. There are two kinds of swimlanes: *Pools* and *Lanes*. There is a simple relation between the two: a pool is composed of multiple lanes. :

- *Pools:* represent participants in a business process, which can be a specific entity or a role. A pool is in general a container and *regroups several flow objects*, representing the work that the pool needs to perform under the process being modelled. To facilitate the clarity of the business process, a pool will extend the entire length of the diagram, either horizontally or vertically. There is no specific restriction to the size or positioning of a pool. Also, a pool acts as the container for the sequence flow between activities of a business process. The sequence flow can thus cross the boundaries between the different lanes of a pool, but cannot cross the boundaries of a pool. To represent the interaction between several pools the message flow is used. All business process fragments contain at least one pool. In most cases, is the diagram consists of a single pool, it will only display the activities of the process fragment and may omit to display the boundaries of the pool.
- *Lanes:* are sub-partition of pools. As with pools, they can be used to represent specific entities or roles involved in the process fragment. They extend the entire length of the pool, either vertically or horizontally. Lanes are mainly used to organize and categorize the activities within a pool.

The last category of elements in BPMN are called *artifacts*. They are used to provide additional information about the process. There are three artifacts specified in BPMN but modelling tools are allowed to add as many new artifacts as they need:

- *Group:* grouping of activities does not affect the Sequence Flow. The grouping can be used for documentation or analysis purposes.
- *Text Annotation:* are a mechanism for a modeller to provide additional information for the reader of a BPMN Diagram.
- *Data object:* are considered artifacts because they do not have any direct effect on the sequence flow or message flow of the process, but they do provide information about what activities require to be performed and/or what they produce.

Figure 2.3.3 depicts the subset of the most commonly used BPMN elements and how they are graphically represented.

The BPMN specification defines the Business Process Diagram modelling objects, already presented, but also the semantics of their behaviour. The BPMN 2.0 specification document [OMG11] provides an entire section dedicated to defining the execution semantics of BPMN processes. The purpose of this execution semantics is to describe a clear and precise understanding of the operation of BPMN elements. The execution semantics are described informally (textually).

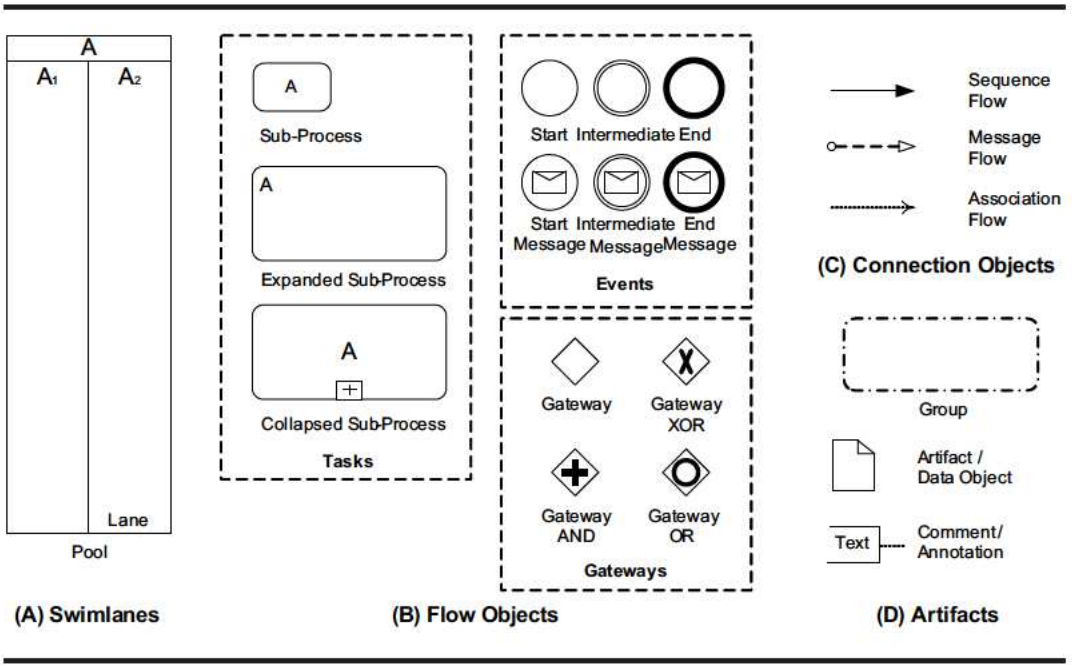


Fig. 2.6: Graphical notation for core set of BPMN elements

To facilitate the definition of process elements behaviour, the standard employs the concept of a *token* that will traverse the sequence flows and pass through the elements in the process. A token is a theoretical concept that is used as an aid to define the behaviour of a process that is being performed. The behaviour of process elements can be defined by describing how they interact with a token as it "traverses" the structure of the process. However, modelling and execution tools that implement BPMN are not required to implement any form of token.

A Process is instantiated when one of its start events occurs. Each start event that occurs creates a *token* on its outgoing sequence flows, which is followed as described by the semantics of the other process elements. A Process instance is completed if there is no token remaining within the process instance. For a process instance to become completed, all tokens in that instance must reach an end node, where the tokens are "consumed".

As presented, the execution semantics of BPMN is defined in terms of enabling and firing of elements, based on a token-game. The dynamic behaviour of *Petri Nets* is also defined in terms of firing of transitions, triggering the passing of a token through the net. Petri Nets, one of the best known process modelling languages, is presented in the following section.

2.3.4 Petri Nets

Petri nets (PN) [Pet77] are a formal, mathematical framework for the modelling and analysis of complex concurrent systems. Petri nets were originally formalised by Carl Adam Petri in his thesis in 1962 [Pet62] at the Technische University Darmstadt, Germany. Since their emergence, they have been extensively studied and a recent bibliographical study shows that there are currently over 8500 papers in the Petri Net literature [Wor12].

Petri nets have their foundation in theoretical computing science with numerous papers

being published on their fundamental concepts, theory and algorithms. Due to their popularity, they have been intensively applied in a wide range of disciplines, including manufacturing, hardware design, business process management, verification of distributed algorithms, local area networks, neural networks and more recently biological networks [DRR04]. Their popularity is due to both the easy-to-understand graphical representation and their potential as a technique for formally analysing concurrent systems.

2.3.4.1 Place Transition Nets

A Petri net is a directed bipartite graph, in which the nodes represent *transitions* (discrete events that may occur), *places* (conditions) and *directed arcs* (describe which places are pre- and/or post-conditions for which transitions). They have a simple graph-based representation.

Transitions are active components. They model activities which can occur, thus changing the state of the system. Transitions are only allowed to fire if they are enabled, which means that all the preconditions for the activity have been fulfilled. Transitions are graphically represented as rectangles. Places in a Petri net model local system states and are represented as empty circles. State changes are modelled by the flow relation which connects places with transitions and transitions with places using directed arcs. An arc can connect a place to a transition, or a transition to a place, but is not permitted to connect a place to a place, or a transition to another transition. An arc is given a weight which signifies the replication of that arc. The connectivity of the places and transitions gives the structure, or topology of the net.

Each place in a Petri net may contain a non-negative number of *tokens*. Such a token is graphically represented by a black dot. Tokens denote the number of the particular resource contained by the place. The number of tokens in a particular place is called the *place's marking*. A distribution of tokens over the places of a net is called a *net marking*. The state of a Petri net is determined by the number of tokens present in each place. The initial state of the system is represented by the *initial marking*.

A transition is said to be *enabled* in a given marking if all of its input places have a marking greater than the weight of the arc from the place to the transition. Enabled transitions can *fire*, which represents an event, or reaction happening, and alters the marking of the net. The firing of a transition changes the marking of the net (state of the system), consuming a number of tokens (equal to the weight of each input arc) from each of its input place, and producing an amount of tokens (equal to the weight of each output arc) to each of its output place. Transitions do this atomically, in one non-interruptible step. The interactive firing of transitions in subsequent markings is called *token game*.

The class of Petri nets described above is called *place/transition nets (PT nets)* and it is the most general one.

Definition: a place/transition net PN is a tuple, $PN = \{P, T, F, W, M_0\}$, where:

- $P = \{p_1, p_2, ..., p_n\}$ is a finite set of places names;
- $T = \{t_1, t_2, ..., t_m\}$ is a finite set of transitions names, such that P and T are disjoint: $P \cap T = \oslash$;

- $F \subseteq (P \times T) \cup (T \times P)$ is the flow relation (arcs), where $(x, y) \in F$ denotes an arc from x to y;
- $W : F \to \mathbb{N}^*$ is the weight function;
- $M_0 : P \to \mathbb{N}$ is the initial marking of the Petri net.

One of the important aspects that makes Petri nets interesting is that they provide a balance between modelling power and possibility of analysis: many properties one would like to know about concurrent systems can be automatically determined for Petri nets, although some of them are very expensive to determine in the general case. There are several behavioural properties of Petri nets [Mur89]:

- *Boundedness:* this property tells us how many (and which) tokens a place may hold when we consider all reachable markings. A Petri net is said to be *k-bounded* if the number of tokens in each place does not exceed a finite number k, for any marking reachable from the initial marking. For a particular place of the net, we have the *best upper bound* (maximal number of tokens that can reside on a place in any reachable marking) and the *best lower bound* (minimal number of tokens that can reside on a place in any reachable marking). Finally, a net is said to be *safe* if it is 1-bounded. By verifying that the boundedness or safeness properties are respected that there will be no overflow in the buffers or registers, no matter of the firing sequences taken.
- *Reachability:* is one of the fundamental properties that can be studied for Petri nets. The reachability problem for Petri nets is to decide, given a net N and a marking M, if that marking can be reached from the initial marking M_0. A marking M is said to be reachable if there is a firing sequence that transforms M_0 into M. This is just a matter of traversing the reachability graph until either we reach the requested marking or we know it can no longer be found. However, this is harder than it may seem at first: the reachability graph is generally infinite, and it is not easy to determine when it is safe to stop. It has been shown that the reachability problem is decidable although it takes at least exponential space and time to verify in the general case.
- *Liveness:* this concept is closely related to that of complete absence of deadlocks in operating systems. a Petri net can be described as having for degrees of liveness $L_1 - L_4$. This depends on the liveness of its transitions, where a *live transition* is characterized by the fact that from any marking there exists an occurrence sequence which can enable this transition. Thus a Petri net is said to be live if all of its transitions are live, or in other words if, no matter what marking has been reached form the initial marking M_0, it is possible to ultimately fire any transition of the net. This property ensures that a net is deadlock-free no matter what firing sequence is chosen. The property also allows to identify unwanted infinite firing sequences, verify that a transition can alwwys re-enabled and ensure that all transitions of the net will ultimately get enabled.
- *Home properties:* tell us about markings (or sets of markings) to which it is always possible to return. The home properties tell us that there exists a single home marking M_{home}, which can be reached from any reachable marking. This means that it is impossible to have an occurrence sequence which cannot be extended to reach M_{home}. In other words, we cannot do things which will make it impossible to reach M_{home} afterwards. The set of all home marking of a net is called the *home space*.

- *Dead markings:* are markings in which no transitions are enabled. The verification of existence of dead markings allows to check that the net does not run into unwanted deadlocks.
- *Fairness:* tell us how often individual transitions occur. This property is only relevant if there are Infinite Firing Sequences(IFS). Given a transition t it is often desirable that t appears infinitely often in an IFS. The following situations might occur:
  - t is impartial: t occurs infinitely often in every IFS;
  - t is fair: t occurs infinitely often in every IFS where t is enabled infinitely often;
  - t is just: t occurs infinitely often in every IFS where t is continuously enabled from some point onward;
  - No fairness: not just, i.e., there is an IFS where t is continuously enabled from some point onward and does not fire any more.

    The notions presented above concern more the Petri net simulator semantics rather than the Petri net semantics itself.

There are other, less important behavioural properties of a Petri net, which we only mention: *coverability*, *persistence* and *synchronic distance* [Mur89].

Petri nets have a number of desirable features as a modelling framework:

- a clear graphical visualisation of the system;
- the ability to analyse the network topology and to determine structural properties;
- the ability to analyse the network topology and state to determine behavioural properties;
- the ability to simulate a network by the addition of a time element;
- the ability to analyse a model with high level variants of a Petri net.

Due to their popularity, a lot of Petri net formalisms have emerged. In [BDC92], Bernardinello et al. propose a possible classification that distinguishes between three basic net levels:

- *Level 1:* are characterized by "boolean tokens", i.e. places are marked by at most 1 unstructured token.
  - *Condition/Event Systems:* are Petri Net Systems of level 1 which require the net structure to be pure and simple. The semantics are defined by the full marking class (forward and backward reachability) and require 1-liveness.
  - *Elementary Net Systems:* have been defined by G. Rozenberg and P. Thiagarajan as a more simpler model with only forward reachability.
  - *1-safe systems:* belong to net systems of level 1, since their places are marked by at most one unstructured token.
- *Level 2:* are characterized by "integer tokens", i.e. places are marked by several unstructured tokens - they represent counters.

  - *Place/Transition Nets:* are Petri Nets of level 2 characterized by counter tokens, arc weights and place capacities.
  - *Ordinary Petri Nets:* are defined as a subclass of P/T Systems with infinite place capacities and unary arc weights.

- *Level 3:* are characterized by high-level tokens, i.e. places are marked by structured tokens where information is attached to them.
  - *High-Level Petri Nets with Abstaract Data Types:* are high-level algebraic Petri Nets, where the tokens and firing rule are specified over an algebraic specification.
  - *Environment Relationship Nets:* are high-level Petri Nets, where the tokens are environments.
  - *Well-Formed Nets:* are high-level Petri Nets similar to Coloured Petri Nets where the colour functions are defined in a different way.
  - *Traditional High-Level Petri Nets:* were defined as Predicate/Transition nets by H. J. Genrich and K. Lautenbach and as Coloured Petri Nets by K. Jensen.

Important extensions to this general Petri Nets classification are:

- *Colored Petri Nets:* add the possibility to use data types and complex data manipulation. Each token has attached a data value called the token colour. The token colours can be investigated and modified by the occurring transitions. It is also possible to make hierarchical descriptions.
- *Timed Petri nets:* associate with each arc an interval (or bag of intervals). Each token has an age. This age is initially set to a value belonging to the interval of the arc which has produced it or set to zero if it belongs to the initial marking. Afterwards, ages of tokens evolve synchronously with time. A transition may be fired if tokens with age belonging to the intervals of its input arcs may be found in the current configuration.
- *Modular Petri nets:* introduce structure by letting modules be specified separately. The modules communicate either by using shared transitions or place fusion.

2.3.4.2 Hierarchical Coloured Petri Nets

Coloured Petri nets (CPN) [Jen92, Jen94] represent an extension of Place/Transition Petri nets, as they attempt to distinguish individual tokens of PT-nets by giving them colours [Pet80].

When modelling a system by means of a classical Petri nets, elements of this system should represented as tokens, places, transitions and connections. Tokens can be used for modelling physical objects, information objects, collections of objects, states and conditions. In classical Petri nets, however, it is not possible to describe the attributes of a token. It is therefore natural to extend classical Petri nets in such a way that every token carries some data. In a Coloured Petri nets, every place has a type and every token has a value and the value of the token is also called its *colour*. The value of a token can be used to keep up-to-date with information about the object represented by the token [JK09].

To give an overall picture, CP-nets are different from PT-nets in the following ways:

- Tokens in CP-nets can have an arbitrary abstract data type. The data type is referred to as its *colour set* and the value of the token is called its *colour*. All PT-net tokens, in contrast, are of a single, unstructured type.
- Arcs in CP-nets have *arc expressions* associated with them (instead of the simple arc weights of PT-nets). The arc expressions can contain *variables* whose scope covers all arcs associated with a particular transition. The expressions must evaluate to a multi-set of tokens of the appropriate data type for the place associated with that arc.
- Transitions can have guard expressions associated with them. A transition is not enabled unless the guard evaluates to "TRUE".
- Places in CP-nets do not have weights associated with them.
- A marking of the CPN model is given by the number of tokens and the token colours on the individual places which together represent the state of the system.

In a CPN, the enabling and occurrence of transitions happens in the following manner. The arc expressions on the input arcs of a transition together with the tokens on the input places determine whether the transition is enabled, so it is able to occur in a given marking. For a transition to be enabled, it must be possible to find a binding of the variables that appear in the surrounding arc expressions of the transition such that the arc expression of each input arc evaluates to a multi-set of token colours that is present on the corresponding input place. When the transition occurs with a given binding, it removes from each input place the multi-set of token colours to which the corresponding input arc expression evaluates. Similarly, it adds to each output place the multi-set of token colours to which the expression on the corresponding output arc evaluates.

An important extension to CPN is the introduction of *modules* and therefore the possibility to make *hierarchical descriptions*. A CPN model can be organised as a set of modules, in a way similar to that in which programs are organised into modules. There are several reasons why modules are needed. Firstly, creating a CPN model of a large system as a single net is very difficult and impractical, since it would become very large and inconvenient. Secondly, the use of modules as elements of abstraction allow the modeller to concentrate on only a few details at a time. Therefore, CPN modules can be seen as "black boxes", where modellers, when appropriate, can forget about the details within modules. This enables the work at different abstraction levels. Finally, there are often system components that are used repeatedly. A module can be defined once and used repeatedly, so it also serves as a unit of reuse.

A hierarchical CPN model consists of a set of modules (pages) which each contain a network of places, transitions and arcs. The modules interact with each other through a set of well defined *interfaces*. A page may contain one ore more *substitution transitions*. Each substitution transition is related to a page, i.e., a *subnet* providing a more detailed description than the transition itself. There is a well-defined *interface* between a substitution transition and its sub-page. The places surrounding the substitution transition are *socket places*. The sub-page contains a number of *port places*. Then, socket places are related to port places in a similar way as actual parameters are related to formal parameters in a procedure call. This representation makes it easy to see the basic structure of a complex CPN model and understand how the individual processes interact with each other.

The Petri net research literature proposes three main types of analysis approaches that can be applied on HCPN. Each can be applied to both classical and high-level coloured Petri nets:

- *Simulation:* is the most widely used analysis technique. From a technical point of view, the simulation of a Petri net involves just a "walk" in the reachability graph. By performing several such "walks" it is possible to make reliable statements about different dynamic properties or performance indicators. This technique is mainly used for validation and performance analysis purposes, but cannot be used to prove correctness. Usually, performing a single run does not provide information about reliability of results. Therefore, multiple runs or one run cut into parts: sub-runs. The simulation of HCPN models has many similarities with debugging of programs written in high-level languages such as Java or C. HCPN tools provide different modes of simulation suitable for different purposes. In general, there are two different types of simulations that can be performed:
  - *Interactive:* a HCPN can be investigated and debugged by means of the HCPN simulator, just as a programmer tests and debugs new parts of a program. In the interactive mode of the simulation the user is in full control, sets breakpoints, chooses between enabled binding elements, changes markings of places, and studies the token game in detail. The modeller is able to inspect all details of the markings reached and can see the set of enabled transitions and select the binding elements to occur.The purpose is to see whether the individual net components work as expected. Interactive simulations are, by nature, very slow, as no human being can investigate more than a few markings per minute. Interactive simulations do not require the model to be complete, so the user can start investigating the behaviour of parts of a model and directly apply the insight gained to the ongoing design activities.
  - *Automatic:* is useful later on in a modelling process, when the focus shifts from the individual transitions to the overall behaviour of the full model. This type of simulation allow us to obtain much faster simulations. A totally automatic simulation is executed with a speed of several thousand steps per second (depending on the nature of the CPN model and the power of the computer on which the CPN simulator runs). The user is in control of automatic simulations by means of stop options which make it possible to give an upper limit to the number of steps the simulation should run.

- *Place/Transition invariants:* this concept can be used to partially address the problem of state space explosion that appears in Petri net analysis. *Invariants* define net properties independent of the initial state of the Petri net. Such invariants may be applied to either places or transitions. Invariants can be computed using linear algebraic techniques.

- *State-space analysis:* simulation can only be used to consider a finite number of executions of the model being analysed. This makes it likely that the protocol works correctly, but it cannot be used to ensure this with absolute certainty since we cannot guarantee that the simulations cover all possible executions. The state space of a HCPN is a directed graph with a node for each reachable state and an arc for each possible state change. Full state spaces represent all possible executions of the model

being analysed. The basic idea is to calculate all reachable states (markings) and all state changes of the HCPN model and represent these in a directed graph. The state space of a HCPN model can be computed fully automatically and makes it possible to automatically verify that the model possesses an abundance of properties, like reachability, boundedness, liveness, and fairness.

State spaces are also referred to as *occurrence graphs* or *reachability graphs/trees*. The term *occurrence graph* denotes the fact that a state space contains all the possible occurrence sequences of the HCPN. The *reachability graph/tree* is used because the state space contains all reachable markings of the net.

One of the main drawbacks of the state-space analysis approach is the *state space explosion problem*, which might limit the application of this method to very large HCPNs. For large models the state space is often so huge that it cannot be fully generated. Therefore, the user has to focus only on certain aspects of the model and generate only a sub-graph of the state space. As the aim of the state space models is to analyse the overall behaviour of the HCPN, standard queries may be applied for determining certain behavioural properties.

3. SPL METHODOLOGY FOR THE DERIVATION OF PRODUCT BEHAVIOUR

Abstract

Throughout this chapter, we propose a new software product line engineering methodology that focuses on the derivation of product behaviour. A methodology can be seen as a framework for applying software engineering practices with the specific aim of providing the necessary means for developing software-intensive systems. By applying the proposed methodology, behavioural product models can be produced that belong to the analysis and early design levels of the software development life-cycle. We focus on behavioural models as this type of product representation is currently not sufficiently addressed in product line engineering. The behavioural models obtained should describe the business and operational step-by-step work flows of activities/actions performed by the derived product. The main flow of the methodology and its specific steps are described in Section 3.1. The first step of the methodology, described in Section3.2, focuses on capturing the common aspects and those that discriminate among systems in the product family using feature models. The second phase of the methodology focuses on the creation of business process fragments, which represent the core assets of the software product line, and is presented in Section 3.3. Throughout Section 3.4 we briefly discuss the concept of "correctness" for business process fragments and explain what type of verifications are required to ensure this property. Section 3.5 aims at bridging the gap between feature models and solution models and thus defines a mapping of features to model fragments specifying the concrete feature realisations. Section 3.6 is the first one that belongs to the Application Engineering phase. It consists of selecting, based on the user's preferences, the required features that will be part of a particular product that is derived. Finally, in Section 3.7, the set of business process fragments resulting from the feature diagram configuration step are transformed, through a compositional approach, into a proper business process that models the behaviour of the SPL product being derived.

Software development is a complex and tedious task. As a consequence, software engineers are unable to produce complex and high-quality applications in an ad-hoc manner. *Methodologies* are the means provided by software engineering to facilitate the process of developing software and, as a result, to increase the quality of software products.

Methodologies have been successfully applied in various disciplines and domains for a long time, before their introduction to software engineering. A methodology usually provides guidelines for solving a problem, with specific components such as phases, tasks, methods, techniques and tools [IR05]. Jayaratna emphasises that a methodology provides an explicit way of structuring systems development: *"Methodologies contain models and reflect*

particular perspectives of 'reality' based on a set of philosophical paradigms. A methodology should tell you 'what' steps to take and 'how' to perform those steps but most importantly the reasons 'why' those steps should be taken, in that particular order" [Jay94]. The concept of *"methodology"* may also be defined as follows [MW12]:

- *"the analysis of the principles of methods, rules, and postulates employed by a discipline"*;
- *"the systematic study of methods that are, can be, or have been applied within a discipline"*;
- *"the study or description of methods"*.

In the field of software engineering, a methodology provides a structured set of guidelines, methods, descriptions and tools for each phase in the life cycle of a system, to ensure the production and maintenance of a well-engineered product that is fitted for its purpose. It can also be seen as a framework for applying software engineering practices with the specific aim of providing the necessary means for developing software-intensive systems. Methodologies may differ widely in terms of their philosophy, objectives and system modelling approaches. Therefore, several authors have defined this concept in the context of software engineering:

- Avison et al. state that *"a methodology is a collection of procedures, techniques, tools and documentation aids which will help the systems developers in their efforts to implement a new information system. A methodology will consist of phases, themselves consisting of sub-phases, which will guide the system developers in their choice of the techniques that might be appropriate at each stage of the project and also help them plan, manage, control and evaluate information systems projects"* [DEA03];
- Maddison et al. define a methodology as *"a recommended collection of philosophies, phases, procedures, rules, techniques, tools, documentation, management and training for developers of information systems"* [MBB+84].

A methodology therefore permit individuals to structure their understanding of appropriate solutions for a problem situation, according to their perspective and their previous experience of both the problem context and the methodology. It affects the way in which individuals will perceive the context and tasks of development, with each component layer of the methodology acting as a filter to the next layer. Ultimately, the problem situation is perceived through the filters provided by successive elements of the methodology.

Moreover, software engineering methodologies have been demonstrated to be important in two respects:

- Facilitate standardisation and thus manage the development process, decreasing individuals' autonomy and discretion in design decisions;
- Embody the values of technical development staff reinforcing and propagating those values through the normative processes of design.

SPLs have recently been introduced as one of the most promising advances for efficient software development. This technique has gained a lot of attention in recent years by both research and industry. Throughout the past years, the SPL comunity has mainly focused on the domain engineering phase of the process. Application engineering, or product derivation, a key phase of the SPL process that can be tedious and error-prone [DSB05], has been given far less attention compared to domain engineering. Implicitly, there arises the need for new product derivation techniques in the SPL research field.

To address this situation, SPLE has recently turned towards Model-Driven Engineering, identified as a software development paradigm able to offer viable solutions for improving product derivation. In this context, the result of the derivation process is the model of an individual product obtained from the set of core assets. Two types of models, each offering a different view of the derived product, can be obtained: *structural* and *behavioural* [RC10]. Structural models provide a static view of the derived product. Behavioural models illustrate the dynamic behaviour of the product and the general flow of control. Most of the work in SPLE addresses the derivation of structural product representations, neglecting or just briefly addressing the problems inherent to the derivation of product behaviour. This yields an unwanted situation, as the behavioural product representation is as important as the structural one.

Moreover, upon closer examination, there are only few guidelines or methodologies available that try to address to some extent the issue of deriving product behaviour. These approaches typically focus on enhancing specific steps of the SPLE development, without being a methodology that covers the SPLE process end-to-end.

Our goal in this section is to introduce and define a SPL engineering methodology which allows the development and the derivation of behavioural models of SPL products. The methodology covers the entire SPLE process, from variability modelling and core assets definition during the domain engineering step all the way to the actual product derivation during application engineering. We focus on behavioural models as this type of product representation is currently not sufficiently addressed in product line engineering. The behavioural models obtained should describe the business and operational step-by-step workflows of activities/actions performed by the derived product. We propose a process-based language called Composable Business Process Fragments (CBPF), based on the BPMN standard, as the specific type of model used for representing the behaviour of derived products. However, the methodology is generic and can also work with other types of behavioural models. The methodology is intended to support both domain engineering and application engineering phases of SPL software development process.

3.1 Overview of the methodology

We propose a new software product line engineering methodology that focuses on the derivation of product behaviour. By applying this methodology, we can produce behavioural product models that belong to the analysis and early design levels of the software development life-cycle. The proposed methodology covers only the derivation of behavioural product models, and does not address the structural product representation. However, it can be used together with other product derivation techniques aimed at obtaining structural product models.

We present in the following the *key features*, or the "meta-requirements", desired for this

methodology:

- The methodology *adopts and follows the traditional software product line engineering approach* by splitting the overall development life-cycle into two phases: domain engineering (DE) and application engineering (AE). From this point of view, the proposed methodology can be considered *"complete"* or *"end-to-end"*, covering the entire SPLE process. During domain engineering, the focus is on core assets development. For this methodology, the core assets created are *business process fragments*. Moreover, during this phase of the process, we also address the variability of the product line. Feature models are used for capturing product line commonality and variability. During the application engineering phase, new behavioural product models are created from the core assets base. Based on a user guided selection, a set of business process fragments are selected. A new business process, modelling the behaviour of the desired derived product, is then obtained following a compositional approach.

- The methodology *follows the separation of concerns principle*. In Section 2.1.6 we introduced a classification of variability modelling approaches. In our methodology, we use a variability modelling technique that belongs to the second class of approaches, those that *distinguish and keep separate the assets model from the variability model*. More precisely, we use feature diagrams as a means to capture the product line variability. Features are then connected to business process fragments, which represent the core assets. Therefore, a clear separation of concerns is achieved. Some implicit advantages are: each asset model may have more than one variability model; designers can focus on the product line itself and not on its variability, which is addressed separately; possibility for a standardized variability model.

- The methodology makes use of *positive variability* [VG07]. In MDE-based SPLE, models are used to represent products in the problem and solution domain. Consequently, a solution domain model often needs to be adapted based on a product configuration in the problem domain. In other words, we want to use a configuration of the variability model, in our case the feature model, to guide the derivation of the desired product model. There are two fundamentally different ways of approaching this problem: use *negative variability* or *positive variability*. *Negative variability* selectively takes away parts of a creative-construction model based on the presence or absence of features in the configuration model. The "overall" model is built manually, and model elements in that model are connected to features in the configuration model. With *positive variability* we start with a minimal set of core assets and progressively add additional parts. In our methodology, we use positive variability in the following manner: a configuration of the feature diagram is created based on a specific user selection; implicitly, business process fragments connected to those features are also selected; the fragments are then combined using a compositional approach into the resulting behavioural product model.

- The methodology allows *to model complex product behaviours using a compositional approach*. Since the methodology focuses on behavioural representations of SPL products, an important aspect to be addressed is how to obtain a complex behaviour from several simpler ones? One of the factors that contributes to the difficulty of developing complex behaviours is the need to address multiple concerns in the same artefact. This situation emphasizes the need for separation of concerns mechanisms

as a support to the design of complex behaviours, modelled using business processes in this case: concerns are defined separately, and assembled into a final system using a compositional techniques. This challenge is pointed out by Mosser: *"there is no approach described in the literature which fulfils the specific goal of supporting the design of complex business processes following a compositional approach, at model level"* [Mos10]. The use of the separation of concerns principle in this methodology was explained in the previous paragraphs. The methodology uses a *compositional approach* for creating the final derived products. The same compositional approach can also be used for creating individual business process fragments. For this purpose, we propose a set of business process composition operators that are applied on the business process fragments and produce the final behavioural product model.

- The methodology uses the notion of *"composable business process fragment"* and provides language support for it. *Business process fragments* are introduced as a new unit of reuse for business process modelling. They fulfil the need of another unit of reuse, one that allows fine-grained reuse of process logic within the range from atomic language constructs to sub-processes and whole processes. We also propose a language that supports the modelling of such business process fragments. To address the issue of process composition, our language proposes the concept of *"composition interface"* and *"composition tag"*. Using an annotation-based mechanism, composition interfaces are used to explicitly identify the parts of a process fragment where it can connect to other fragments or where other fragments can be connected to it. The interfaces are also an indicator of how this connection can be performed.

- The methodology *ensures several desirable structural and behavioural correctness properties are fulfilled through a Petri nets based verification.* We define the notion of *"correctness"* for composable business process fragments in terms of *structural and behavioural correctness.* To ensure *structural correctness*, several well-formedness properties are defined on the business process fragment meta-model. This ensures that those properties are fulfilled for every business process fragment that conforms to the meta-model. Regarding *behavioural correctness*, we ensure that business process fragments satisfy two types of dynamic properties: *predefined* (absence of dead tasks, livelock analysis, deadlock analysis, reachability analysis for end states and composition interfaces) and *specific* (differ from one process to another) properties.

The methodology provides a step-by-step guide to creating behavioural product models. Complete methodologies consist of many steps, each addressing different aspects of the product life cycle. The main flow of the methodology and its specific steps are described in the following.

- **Construction of the feature diagram:** the first step of the methodology consists of *capturing the variability of the SPL using feature models.* They are constructed based on *requirement documents* and *user specifications.* It involves a thorough analysis and parsing of the documents to *extract the features.* Then, the actual *feature diagram construction* is performed: determine variation points and associated variants, define feature groups and relations between them, choose mandatory and optional features, express cross-tree feature dependencies. The end result is the feature diagram of our SPL.

- **Construction of business process fragments:** for our methodology, business process fragments represent the *core assets base* of the product line. There are three possible ways by which they can be created.

  - *Construct form scratch:* new business process fragments can be created as the actual implementations of the features. The product line engineer creates them based on feature descriptions and detailed information from the requirement documents. The specific knowledge of a domain expert in the field is required for obtaining proper business process fragments;
  - *Select from fragment library:* business process fragments might already be available in different forms in *business process libraries/repositories*. In this case, we can simply select a fragment that corresponds to the current requirements, which can be used "as-it-is", without further modifications;
  - *Adapt existing fragment:* business process fragments can be found in process libraries that only partially satisfy the requirements. In this case, such processes can be partially reused and adapted to fit the current needs.

  We provide full language support for creating business process fragments.

- **Verification of business process fragments:** we want to ensure that several properties are fulfilled by the business process fragments created during the previous step. Therefore, each process fragment will be individually verified to guarantee its correctness. We define the notion of *correctness* for business process fragments as the union of *structural correctness* and *behavioural correctness* properties. Structural correctness guarantees that all business process fragments satisfy a series of well-formedness rules. However, we also want to ensure some *dynamic properties*. Two types of properties are checked: *general* ones (should be valid for all business process fragments) like reachability of end events and composition interfaces, deadlock-free process, absence of dead tasks, no infinite occurrence sequences, data type consistency; *process specific* ones (cannot be verified in general and differ form one process to another) for which we propose several general property templates that the user can instantiate for a specific purpose.

- **Associating business process fragments to features:** during the previous steps, the variability of the product line was captured using feature models. Then, business process fragments were created as concrete implementations of the features from the feature diagram. During this step, a *1-to-1 association between features and business process fragments* is performed. Therefore, each feature will have an associated business process fragment that corresponds to its actual implementation. This step needs to be performed manually by the product line engineer, who will assign the business process fragments to the appropriate feature form the feature diagram.

- **Feature diagram configuration:** consists of selecting the features that will be part of a particular product that is derived. The selection of the features is performed according to user requirements. *Configuring the feature diagram* amounts to resolving all the variations it contains. For each variation point, the appropriate variant(s) are selected. Once a particular configuration of the feature diagram has been obtained, the process fragments corresponding to the selected features will also be part of the product configuration.

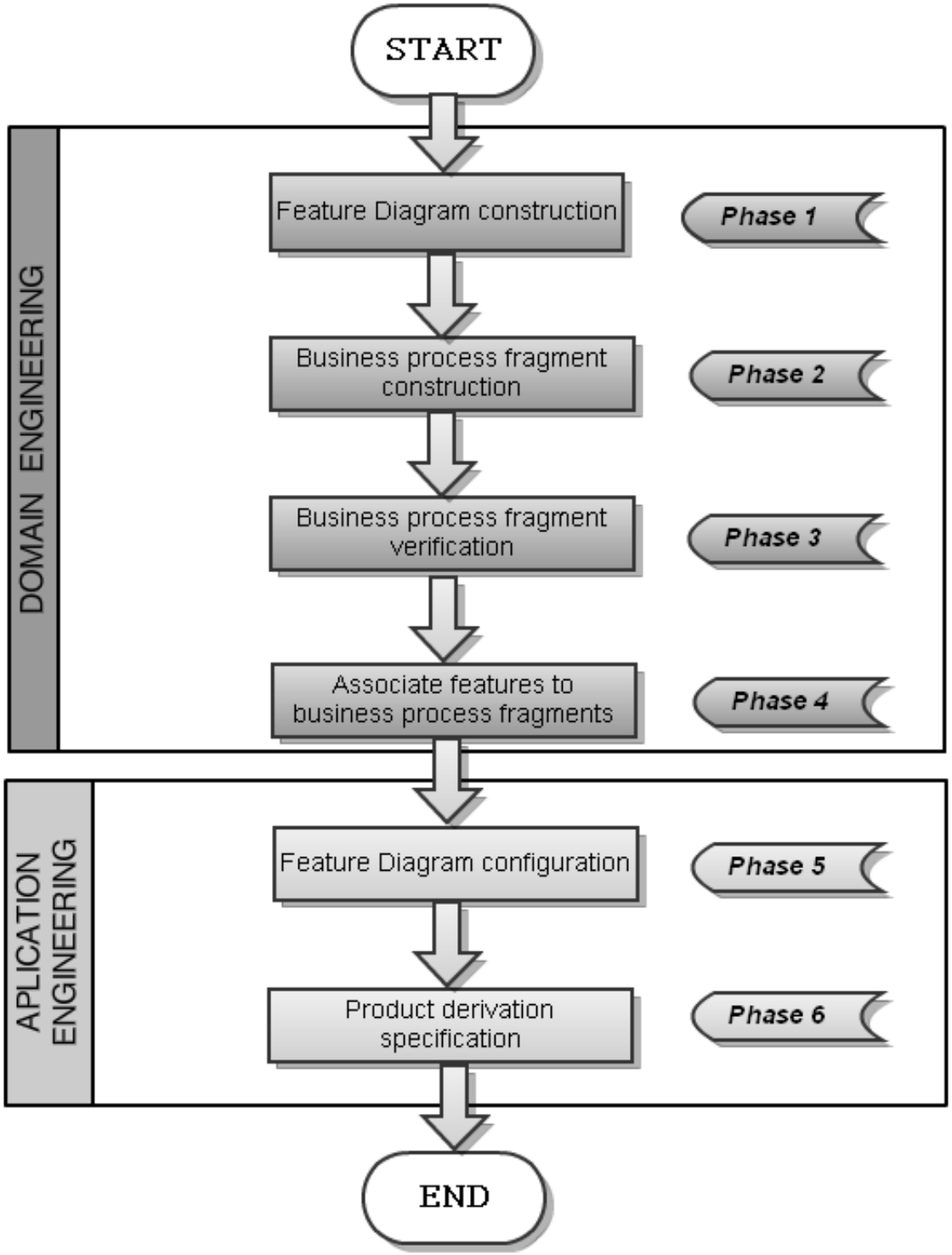


Fig. 3.1: General steps of the proposed methodology

- **Product derivation specification:** during this step of the methodology, the actual business process corresponding to the behavioural model of the derived product is obtained. First, the *set of business process fragments are annotated with composition interfaces*. Their role is to explicitly mark the locations where a business process fragment can be composed with other fragments. Then, the actual order in which the fragments should be composed is specified by means of a composition workflow. The actual composition is a stepwise process: the composition workflow is parsed and for each pair of fragments the corresponding *composition operator* is applied; the result obtained is further composed with the next fragment of the workflow, using the specified operator. The result of the last composition corresponds to the end result of the methodology: the behavioural model of the derived product, represented as a business process model.

From a product line engineering perspective, the first four steps of the methodology belong to the domain engineering process, while the last two steps belong to the application engineering process. A graphical representation of all the steps of the methodology is presented in Figure 3.1. A flowchart is used for the graphical representation. Each rectangle in the diagram corresponds to a distinct step within the methodology and the directed arcs between them indicate how they are related. Steps belonging to domain engineering and those belonging to application engineering are explicitly delimited in the diagram.

3.2 Construction of the feature diagram

Central to the product line paradigm is the modelling and management of variability. The first step of the methodology focuses on defining the system properties relevant to the stakeholders and also on capturing the common aspects and those that discriminate among systems in the product family. To achieve this goal, we use *feature models*, a popular SPL variability modelling technique. A thorough presentation of feature modelling concepts is available in Section 2.1.7.

3.2.1 Feature diagram dialect and meta-model

For the past 22 years, there have been a lot of contributions from research and industry in the area, generating several feature modelling dialects. The feature model used in this thesis is based on previous work from Perrouin [PKGJ08b] and Gouyette [GBLNJ10]. It combines concepts from the FORM [KKL+98a] and Riebisch [Rie03] feature modelling dialects.

FORM is one of the several extensions of the seminal feature modelling approach called FODA [KCH+90]. We keep the main concepts defined in FODA and FORM: there is a *root node*, which refers to the entire system and which is always mandatory. The remaining nodes denote *features* and *sub-features*. Features are *mandatory* by default but can be made *optional*. Features are assembled in *feature groups* and are subject to *decomposition*. Dependencies between features can be expressed using the *mutex* and *requires* constraints. From the FORM approach, we inherit the property that feature names are depicted in boxes. We use FODA and FORM as the basis for our feature modelling language because these notations have the advantage of being clear, well-known, precise and easy to understand.

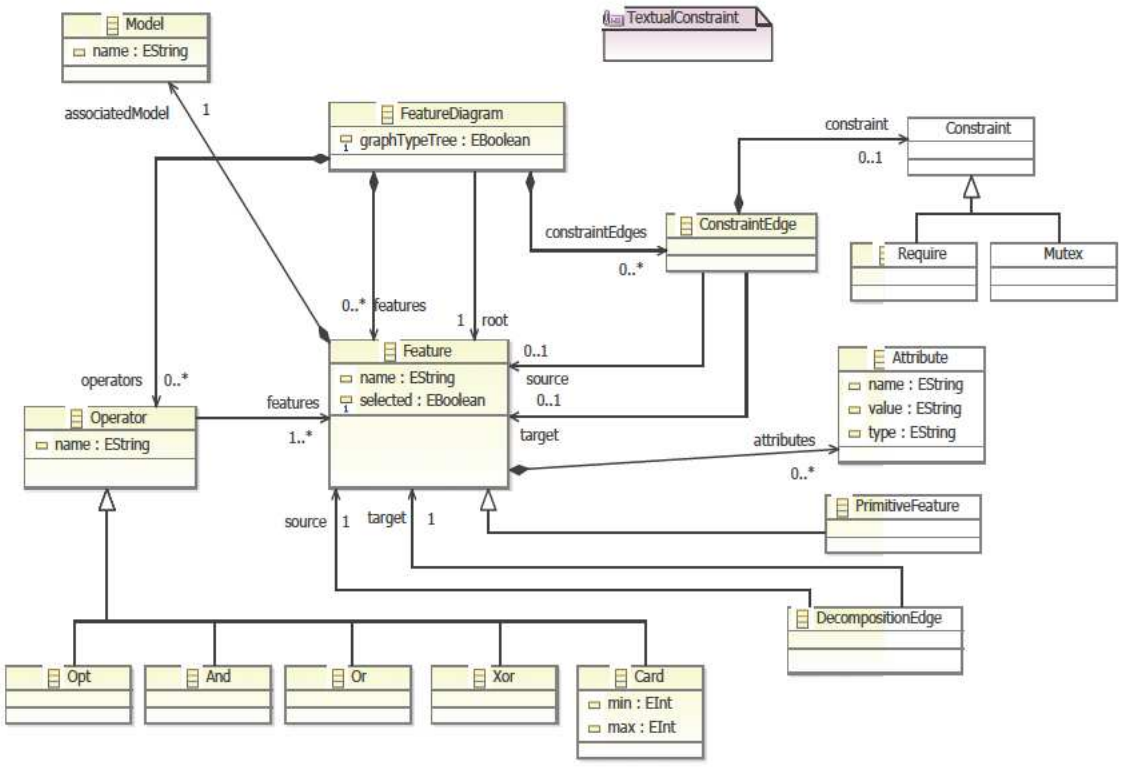


Fig. 3.2: Feature diagram meta-model

From the feature modelling dialect of Riebisch we inherit the concept of *group cardinalities*. Riebisch insists on the importance of representing cardinalities in feature diagrams, and proposes to extend them with UML multiplicities, as the use of alternatives, or and xor relations could lead to ambiguities and only allows multiplicities to be partially represented. We decide to use cardinalities as they confer a great power of expression to the language and allow to represent a large range of possible variabilities.

Following a model-driven engineering approach, the feature modelling language used is presented in the form of a meta-model. The graphical representation is available in Figure 3.2. This meta-model is presented in more detail in the following.

FeatureDiagram is the root class of the meta-model. This class has an attribute *graphTypeTree* which permits to determines whether the feature diagram is a *tree feature diagram* or a *directed acyclic graph* (DAG). It also contains a list of *features* (class *Feature*), which are represented in the feature diagram as nodes. Features can be distinguished by their *name* attribute. It is also possible to express if a feature is part or not of a configuration using the *selected* attribute. There is a unique special *root node*, identified by the reference *root* from *FeatureDiagram* meta-class to the *Feature* meta-class. The notion of *primitive feature* is introduced to distinguish between features that are internal to the feature diagram and the leaf features. A feature is characterized by a set of *attributes*. It can have a *name*, a particular *value* and a specific *type*.

Relations between features can be expressed using *operators*. In the meta-model, these operators are subtypes of the meta-class *Operator*, and each feature (class *Feature*) may contain 0 or more such operators. All the classical feature diagram operators are available.

A feature can be made optional by using the *Opt* operator. There are three operators defined for feature groups: *And*, *Or*, *Xor*. The use of group cardinalities is possible through the *Card* operator. It defines the minimum and maximum number of features that can be selected from a feature group where this operand is applied.

The decomposition of a feature into more refined features is done using *edges* (class *DecompositionEdge*). Edges have features as sources and targets. The set of *constraint edges* is represented in the meta-model by the class *ConstraintEdge* and are contained by the class *FeatureDiagram*. Constraint edges are used to represent cross-tree feature dependency relations. Each constraint edge contains either a *Require* or a *Mutex* constraint.

Moreover, in the meta-model a feature is related to a unique model by the composite association between the class *Feature* and the class *Model*. There is a 1-to-1 association between features and models, specifying that one feature is represented by a unique model.

Finally, feature modelling constraints have been implemented to guarantee the well-formedness of all feature models that are conform to this meta-model. They are defined as a constraints plugin, developed in order to help the user create valid feature models. This plugin is written using Praxis rules [dSMBB10]. The proposed consistency rules are briefly presented in the following:

- *noTwoFeaturesHaveSameName* : a feature can't the same name of another feature in the feature diagram;
- *noParentFeatureAsChildren* : children features cannot contain their parent features;
- *noMutexBetweenParentAndChild* : there cannot be mutual exclusivity between a parent feature and one of its children;
- *noSeveralMutexOnSameFeature* : there cannot exist several mutex constraints between the same pair of features;
- *noCyclesOnRequire* : for two features f_1 and f_2, if f_1 requires f_2, then f_2 cannot require f_1;
- *noBothRequireAndMutexOnSameFeatures* : there cannot exist both mutex and require constraints between the same pair of features;
- *minCardLargerThanZero* : when the cardinality operator is used, the value of the min attribute must be greater or equal than 0;
- *noMinGreaterThanMax* : when the cardinality operator is used, the value of the min attribute must be greater or equal than the value of the max attribute;
- *noMaxLessThanMinusOne* : when the cardinality operator is used, the value of the max attribute must be greater or equal than -1;
- *nbFeaturesMustBeMoreThanMin* : the number of children features of the cardinality operator on a feature must be greater or equal than the minimum cardinality;
- *orOperatorMustHaveAtLeastTwoOperands* : the Or operator must have at least two children features;
- *xorOperatorMustHaveAtLeastTwoOperands* : the Xor operator must have at least two children features;

- *noAncestorFeatureAsChildren* : a child feature cannot have one of his ancestors as children;
- *noConstraintReflexive* : a given feature cannot require itself or be mutually exclusive with itself;
- *noMutexBetweenAndFeatureChildren* : features with the same parent feature and contained in an And operator cannot be mutually exclusive.

3.2.2 Feature diagram construction process

The process is quite difficult due to the fact that the information that needs to be extracted, the domain knowledge, resides in natural language requirements documents and user specifications. The potential size, quantity and heterogeneity of these documents, combined with the inherent ambiguity of natural language, means that the task can be very cumbersome, and can be both time-consuming and error-prone when performed manually. The individual steps of the process followed for constructing the feature diagram of the product line are presented in the following. The entire process for creating the feature diagram is graphically depicted in Figure 3.3.

- **Document analysis and feature extraction:**

  This step of the process relies on and has as input the *requirements documents* and the *user specifications*. It involves the manual or automatic identification and extraction of features. The user or SPL engineer may be faced with a large volume of textual requirements documentation, written in natural language, with all its inherent ambiguity and implicit or tacit knowledge. In addition to this, requirements documents may be heterogeneous, and might include diverse artefacts and concerns from a business as well as a technical standpoint.

  The product line engineer has to parse the requirements document(s) and mine the text in order to identify and extract the features. The context in which this text analysis is performed is quite flexible and can go from single phrases delimited by punctuation marks, to complete paragraph or even subsections. The goal of this analysis is to identify the main requirements and to determine similarity relations them, within one document or across multiple requirements documents. This information can then be used in the process of abstracting them in order to obtain the features of the feature model.

  A large set of requirements are initially extracted from the text. They correspond to an initial possible set of features. However, the size of this initial set will be quite large, so the product line engineer needs to identify similarities between these requirements and then to perform a clustering operation based on such similarities.

  A certain level of flexibility is required in comparing the requirements, not only taking into account words that match, but also taking into account how often they occur together in the rest of the documents: words of requirements occurring together in the rest of the documents suggest that these requirements are similar. Requirements can be considered related if they concern similar matters/subjects. Thus, the subject of the requirements has to be compared, and requirements with similar subjects will be grouped together. Such comparisons are usually performed based on similarity measures. Natural language processing techniques like the Latent Semantic Analysis

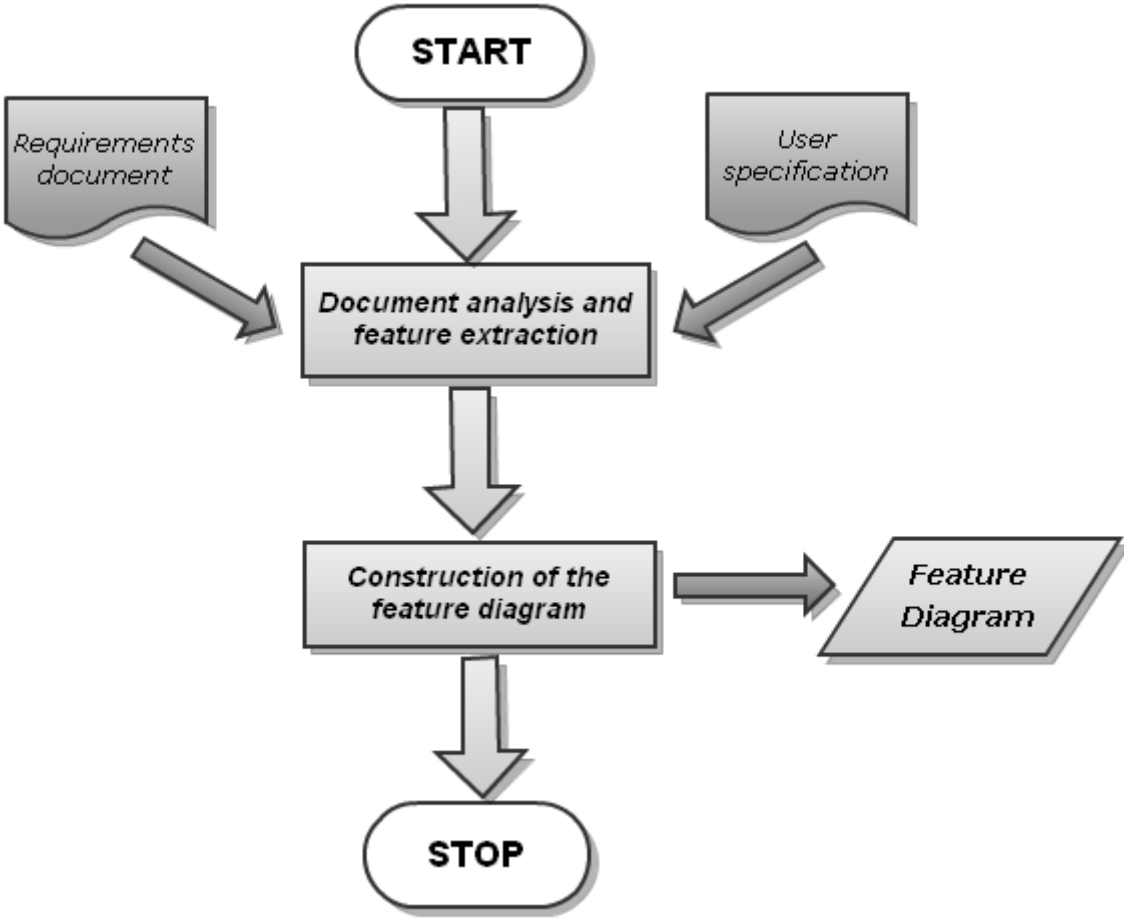


Fig. 3.3: Complete process for creating the feature diagram

(LSA) can also be used. LSA considers texts similar if they contain a significant set of semantically similar terms, where semantic similarity is deduced by examining term distribution among the entire document set.

Following this step, requirements which are found to be semantically similar (have the most in common), are "clustered" to form a feature. This operation is performed based on the intuition that features are tightly-grouped clusters of related requirements. Pre-existing hierarchical relationships between requirements should also be captured. The clustering is an iterative process, so the smaller features are then clustered with other features and requirements to form a larger feature. It is up to the product line engineer when to stop the clustering process and to determine the maximum number of levels in the feature hierarchy.

Ultimately, the result obtained is a hierarchy of features, which are clusters of requirements. A refinement of the resulting features can the be performed, including the manual naming of the features.

- **Creation of the feature diagram:**

  Building feature models is far from a trivial task, even when features have been identified, as the product line engineer will need to distinguish which are the core and which are the variant features, and also to deal with the product line variability. Once the set of features from which the feature model will be created have been selected, there remain several activities to be performed to transform them into the actual SPL feature model.

  - *Identification of core/mandatory features:* mandatory features define the core of the product line. A feature is considered mandatory if it belongs to every possible configuration of the feature model, so it implicitly appears in all the products that can be derived from the SPL. They can be considered as the prerequisites for every product that will be built and are the foundation of the SPL, onto which all the products are built. Mandatory features correspond to key requirements which are essential for the product line. The product line engineer therefore needs to carefully select from the available set of features those he thinks will become the backbone of the SPL.
  - *Identification of optional features:* in a similar manner as for mandatory features, the product line engineer needs to define the features that will appear as optional in the feature diagram. An optional featured defines a requirement or functionality that can be omitted from some of the products of the SPL. They are specific characteristics of some individual product(s). Optional features are not part of the backbone of the SPL. Optionality is one possible way of representing variation if feature diagrams. The selection process of optional features is quite straightforward: all the features that are not defined as being mandatory implicitly become optional.
  - *Identification of variability:* this is a crucial step in the construction of the feature diagram. Variability needs to be first detected by analysing the requirements documents. In order to identify variability already present in these documents, we determine words whose semantic category denotes variability. For this, a variability lexicon can first be defined and then a grammatical pattern identification process applied on the text. The variability lexicon is a collection of words (e.g., different, like, such as, several, and, or) which point to the potential presence of variability elements in the text. Grammatical patterns are

patterns of natural language that denote the potential presence of variability, for instance enumerations. When a variability element is considered relevant for inclusion into the feature model, the analyst needs to decide upon the semantics of the variation. Moreover, the product line engineer needs to consider to which level of the feature model this variability should be propagated. He also needs to consider how to represent this variability in the feature model: what sort of optionality has been revealed, whether sub-features need to be created and so on.

- *Selection of variation points and variants:* a variation point identifies one or more locations at which the variation will occur. They offer the required flexibility and adaptability to the SPL. In this context, variation points define high level requirements for which several possible concrete implementations exist. Based on the results of the variability identification step, the product line engineer needs to select from the feature set those that become variation points. However, variation points can also be created "artificially": if several similar features are identified during the variability identification step, a new higher-level feature that subsumes them can be introduced and becomes a variation point. Once defined, a variation point needs to have several variants. They represent concrete possible implementations of that variation point. Use of variation points and variants is the main variability implementation technique for feature models. As for variation points, variants are also selected based on the results of the variability identification step.
- *Applying variation operators:* the variants connected to a variation point can be grouped together into a feature group. Different variability relations may exist between the variants of such a feature group. Variation rules or operators define the manner in which a variation point is replaced by one or more of its variants. At the variation points, the variation rule/operator is applied according to the semantics of the natural-language operator. The rules specify the semantics of the variation - that is, how the variation manifests itself: the inclusion or removal of sub-features, concerns, or the composition of requirements in new ways. Variation operators like and, or, xor, cardinality are defined and applied. The variation rules/operators are deduced from the results of variability identification.

Constructing a feature diagram from a requirements document is a difficult task, that can become both time-consuming and error-prone when performed manually. Some authors have worked on automating this process. In [WCR09], Weston et al. introduce a tool suite which automatically processes natural-language requirements documents into a candidate feature model, which can be refined by the requirements engineer. The framework also guides the process of identifying variant concerns and their composition with other features. Feature models produced by this framework compare favourably with those produced by domain experts. Acher et al. [ACP+11] try to facilitate the transition from product descriptions expressed in a tabular format to feature models accurately representing them. They propose a process that is parametrized through a dedicated language and high-level directives (e.g., products/features scoping). They also guarantee that the resulting FM represents the set of legal feature combinations supported by the considered products and has a readable tree hierarchy together with variability information.

3.3 Creation of business process fragments

Reuse is a key enabler for improving business efficiency. The rise of BPM techniques and the desire to better manage processes has led to increased awareness and desire to reuse processes. The design of business processes can benefit from reusing existing knowledge. The benefits of reuse have been long recognized and include saving time and resources, reducing development cost, and increasing reliability. Several concepts have been proposed in the field of process-based application development to provide different granularities of reusable process artefacts. There is a need of another *unit of reuse*, which should allow fine-grained reuse of process logic within the range from atomic language constructs to sub-processes and whole processes. The concept of *business process fragment* is a promising candidate to fill this gap. The second phase of the methodology focuses on the creation of business process fragments, which represent the core assets of the software product line. A more detailed presentation of the characteristics of business process fragments and of the necessary steps required to create them are available in the following.

3.3.1 Overview of business process fragments

Today's business dynamics are mandating that business processes be increasingly responsive to change. It is therefore crucial that business process models be modular and flexible, not only for increased modelling agility but also for the greater robustness and flexibility of executing processes. Traditional approaches to business process modelling frequently result in large models that are difficult to change and maintain. Because of their size, these models are not very flexible.

It is important when designing or modelling processes to build in reuse right from the start. To bring more dynamics and flexibility into reuse in business process modelling, we need a more modular and granular way to define and describe reusable parts of business process models. A *business process fragment* is intended as a reusable granule for business process design and can allow for reuse of process logic. This concept is comparable to reusable components in software engineering.

Process fragments represent incomplete process knowledge, which needs to be integrated with further process knowledge to become a complete process model. They are incomplete building blocks containing some local process knowledge that might be useful for more that one business process. By definition process fragment models also have to be composable together into a business process and leave some room for adoptions where the exact business logic is not known at fragment design time. A process fragment represents the implementation of a single abstract activity or functionality. These are the main differences which separate process fragments from classical sub-processes.

To be able to refer to different parts of a process model, several authors have defined process fragments as connected parts of a process model, where boundary nodes of a process fragment can be distinguished as fragment entries and fragment exits based on the directions of incident control flow edges. Defining a process fragment as a connected sub-graph of a process graph is not our intention. We consider business process fragments as self-contained connected process structures which are in most cases created from scratch in a bottom-up approach. Process fragments are designed to implement a set of requirements and model a single abstract functionality, and thus are not a sub-graph of a pre-existing process graph.

Structurally, a business process fragment is a self-contained block of process logic with strictly defined boundaries. Semantically, a process fragment can be addressed as a detailed specification of a high level abstract task or functionality. A process fragment is accepted as a unit of meaningful aggregation of process logic. They need to be coherent and make sense to a domain specialist. Each fragment forms a useful resource in its own right.

There are several characteristics that are required from a business process fragment:

- is a connected process structure with *significantly relaxed completeness and consistency criteria* compared to an executable business process. This is due to the fact that process fragments model partial or incomplete knowledge and are meant to be integrated with other fragments.
- we require business process fragments to be *structurally correct*. This is an important property and is ensured through the definition of several consistency rules, which will be discussed in more detail in Section 4.
- business process fragments are meant to be *composable*. Therefore, a process fragment will contain specific areas where it can connect with other processes.
- has to consist of at least of one start event (entry point) and one end event (exit point). As a process fragment models an abstract functionality, it is required to consist of at least one activity. A business process fragment is *not necessarily directly executable* and it *may be partially undefined*.

There are several ways in which one can interpret a process fragment. We define three process fragment logical viewpoints:

- *Fragment viewpoint:* defines the actual or intended behaviour of the process fragment. This view corresponds to the actual workflow structure of the process, defining the flow of activities. It is the straight-forward way of interpreting a business process fragment.
- *Composition viewpoint:* relates to the fact that process fragments area meant to be composed with other fragments. This viewpoint identifies the *composition interface* of a business process fragment. It specifies the exact places in the fragment where it can be composed with other ones.
- *SPL viewpoint:* defines the behaviour of the fragment as a "black-box", seen from the outside. Process fragments are meant to be concrete implementations of an abstract functionality. Therefore, this viewpoint abstracts from the actual implementation and process structure and focuses only on the functionality (feature) that the business process fragment is meant to implement.

The advantages of process fragments are basically similar to those of code reuse in traditional programming:

- the same logic does not need to be specified over and over again;
- an improved quality of the process design, which can be better assured when the process fragments that are used in the process have an efficient design;

- in case a better fragment is available for a particular task it replaces the less efficient version stored in the repository/library;
- over time, the quality of the process logic that is reused increases with this approach.

To create business process fragments, a domain specific language is required. This language support is one of the contributions of this thesis and will be presented in-depth in Chapter 4. However, in addition to the usage of the language primitives, a process fragment can be created through the composition of two existing fragments by applying a composition operation. We therefore extend the domain specific language for creating business process fragments with a set of composition operators. They are presented in more detail in Chapter 4.

3.3.2 Business process fragment construction process

Business process fragments are the core assets used by the methodology, so their creation is of the utmost importance. The entire process for creating business process fragments is graphically depicted in Figure 3.4. It can be noticed from the figure that there are two main ways in which business process fragments can be created:

- *Construct a new process fragment:* a process fragment can be addressed as a detailed specification of a high level abstract task or functionality. Therefore, new process fragments can be created from scratch as concrete implementations of features from the feature diagram. This construction process is based on the information available in the requirements documents, form where the functional and non-functional requirements for the fragment will be extracted. In most cases, the knowledge and expertise of a domain expert is required and will highly improve the quality of the resulting process fragment. Domain experts are able to identify the key functionalities that the process fragment has to implement and to express this information in a concise, flexible and reusable manner. We promote the idea of business process reuse, therefore newly created process fragments are stored in a business process library/repository, to be possibly used for other software product lines.

  For constructing a new process fragment from scratch, adequate language support is required. We propose a new domain specific language designed specifically for modelling composable business process fragments. The language is based on the BPMN process modelling standard. It only uses a subset of the core BPMN elements, those proven to be the most commonly used when modelling business processes [zMR08]. The choice of elements id detailed in Chapter 4 However, we add new concepts and rules to the language to facilitate the modelling of incomplete process knowledge and to highlight the composability characteristic of process fragments. The use of this language enables the creation of new business process fragments.

- *Reuse existing process fragments:* in this methodology, we promote the idea of business process reuse as it allows saving time and resources, reducing development cost and increasing process reliability. As such, another possible way to obtain business process fragments is by reusing existing ones. For this, a *business process repository/library* is required. The way an organization stores the information about its business processes presents a clue as to whether they are only considered as documentation or true business assets. A Business Process Repository is a central location for

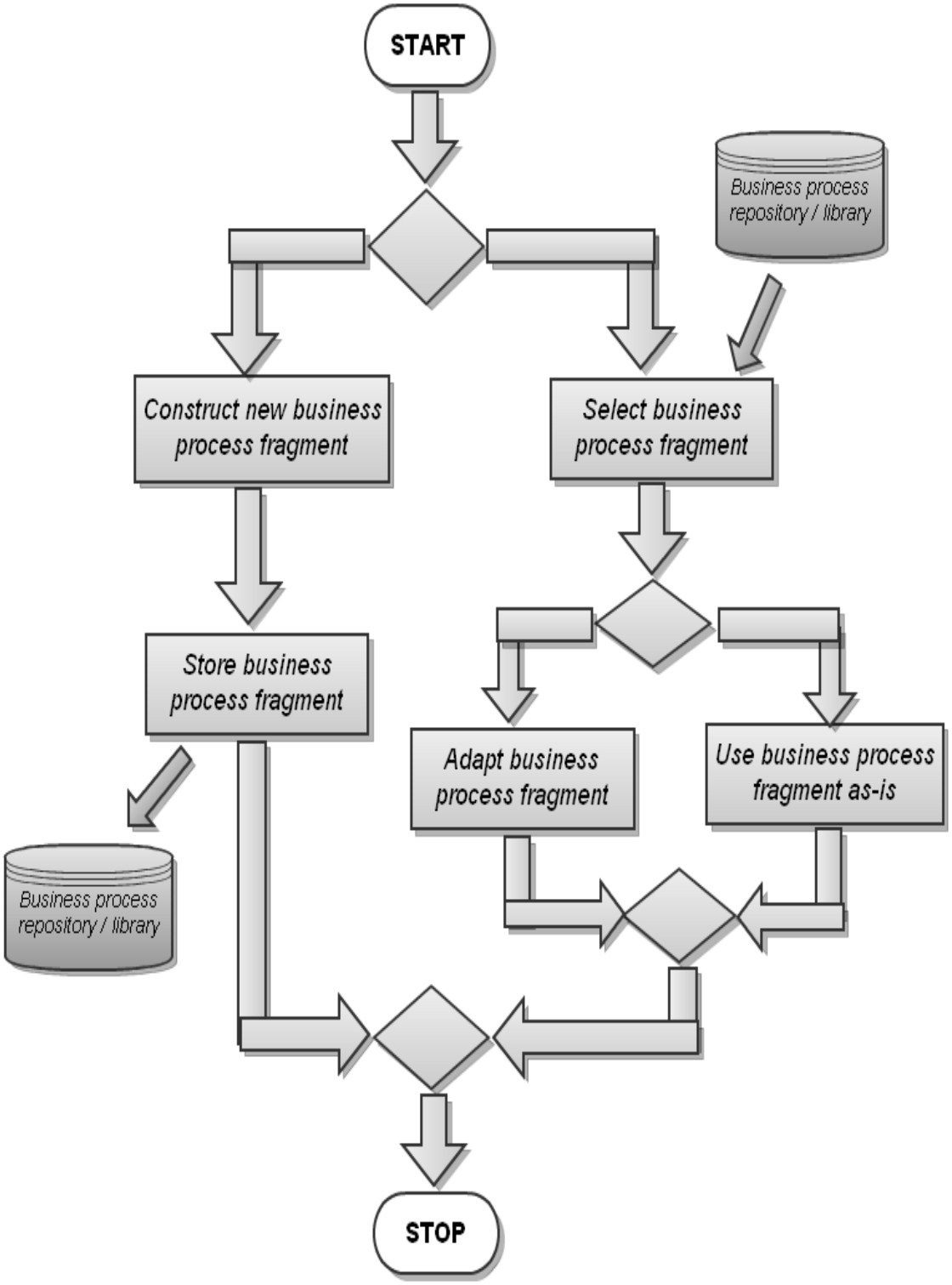


Fig. 3.4: Complete process for creating business process fragments

storing information about how an enterprise operates in the form of business process models. This information may be contained in various storage media, especially electronic. Electronic repositories may range from passive containers for storing process artefacts to sophisticated tools that facilitate such as monitoring, execution, management and reporting on business processes. The availability of a large collection of processes opens up new possibilities, like: extracting knowledge about the operations of the organization from the collection or reusing process fragments from the collection to design new processes.

There are two distinct ways in which a process fragment from a business process repository can be reused in our methodology:

- *Reuse process fragment as-is:* the most straight forward possibility is to directly use the business process fragment, without further modifications or adaptations. This corresponds to an "of the shelf" reuse of process fragments. The product line engineer will select, based on the requirements and the description of the functionality that needs to be implemented, a business process fragments form the process repository that best fits the requirements. The selected process fragments is directly used as-is. Although this case implies a high degree of reuse of process fragments and is a best-case scenario, in real-life product lines it will rarely happen. Due to the specific requirements and particularities of a product line, it is highly improbable that a process fragments that fulfils exactly the requirements can be found or was already developed for another product line.
- *Adapt existing process fragment:* a second possibility is to reuse an existing process fragments by adapting and tailoring it to the specific requirements of the functionality we need to implement. In this case, the product line engineer selects form the business process repository a process fragment that resembles as much as possible and fulfils most of the required characteristics of the process that is being created. The selected process needs then to be slightly modified and adapted so that is entirely models the desired functionality. This type of process reuse is the most probable in real-life projects and product lines.

3.4 Verification of business process fragments

Business process fragment verification is a key phase of the methodology. Verification is concerned with determining, in advance, whether a business process model exhibits certain desirable behaviours. To verify business processes created at analysis and design time is highly desirable. At a late stage of system development process the cost to repair incorrect business processes are extremely high. Therefore, it is reasonable to identify errors at design time. By performing these verifications at design time, it is possible to identify potential problems, and modify the model accordingly before it is used for execution. As the systems created through our product line engineering approach rely on business process models, careful analysis of process fragments at design time can greatly improve the reliability and also allows to correct or optimise the design of such system. Throughout this section we briefly explain the concept of *correctness* for business process fragments and explain what type of verifications are required to ensure this property. As the verification of business process is an important contribution of this thesis, it will be discussed in-depth separately in Chapter 5. Therefore, throughout this section, we only provide an overview of the verification process proposed.

3.4.1 Verification of structural and behavioural correctness

The correctness of models is a major stream of research in business process modelling. Its importance comes from the observation that incorrect process models can lead to wrong decisions regarding a process and to unsatisfactory implementations of the targeted software systems. The verification of business process correctness is essential for ensuring an unambiguous description of the processes. However, the notion of *correct business process* has a wide understanding. By correctness properties, people usually refer to the different kinds of soundness properties [vdAvHtH+11] that have been introduced in the workflow management domain and later on refined.

In this thesis, we define the notion of *correctness* for business process fragments as the summation of two other properties: *structural correctness* and *behavioural correctness*. We discuss each of these properties in the following:

- **Structural correctness:** mainly focuses on avoiding errors at the structural level of business process fragments. In general, structural correctness concerns:
  - the correspondence between the model and the language in which the model is written;
  - the alignment between the model and a set of structural properties that any model of the same type must respect.

  Structural properties refer to the type and number of elements in a process fragments and the control flow relations between them. More precisely, to ensure the structural correctness of a business process fragments, we need to define a set of adequate fragment consistency rules that should be valid for every business process fragment. Following a model driven engineering approach, we define these consistency rules on the business process fragment meta-model. Implicitly, every business process fragment created that is conform to the meta-model will be ensured to satisfy these consistency rules. The rules are defined using the Object Constraint Language (OCL) [OMG06], the standard used for defining constraints on meta-models.

  We propose a set of 26 consistency rules that assure the structural well-formedness of business process fragments. We propose two types of rules:
  - *Based on OMG BPMN specification:* as the business process fragments we propose share a large set of elements with the BPMN language, we consider important to keep the consistency criteria defined by the BPMN standard which are relevant for business process fragments. However, the BPMN documentation does not define well-formedness fo business processes in an explicit and concise manner. This information appears only textually as part of the description and presentation of the different BPMN language elements. Therefore, we needed to extract these rules and express them formally using OCL. The rules defined range form simple ones (business process fragments have exactly one start event, there is at least one end event) to more complex (all flow objects with incoming and outgoing flow relations are on a path from the start event to an end event).
  - *Fragment specific constraints:* as the language for creating business process fragments contains only a subset of the elements of the BPMN standard but also adds new elements and concepts, specific consistency rules are introduced.

They are also expressed using the OCL language. These rules mainly refer to two aspects: the fact that business process fragments model partial information and might be incomplete and the existence of composition interfaces for business process fragments.

- **Behavioural correctness:** structural correctness only allows to check that certain structural properties are valid. However, we also want to perform checks related to the dynamic behaviour of process fragments. Therefore, we define the notion of *behavioural correctness* which serves to verify the possible behaviours of a business process fragment. The concept is defined based on the original definition of *process soundness* proposed by van der Aalst [vdA03] for workflow nets. Behavioural correctness ensures that a business process fragment does not exhibit any erroneous or unwanted behaviours. As the behaviour of a business process fragments is defined by its execution traces, the verification of behavioural correctness is also performed on these traces. However, it should be noticed that this type of verification does not concern the semantic analysis of a business process fragments.

  We propose to verify two kinds of behavioural properties for business process fragments:

  - *Generic:* specify general dynamic properties that any business process fragments should fulfil. Most of them are inspired by the soundness property defined by van der Aalst for workflows and petri nets. The verified properties are:
    - *Reachability of end events:* end events should always be reachable from the start event. This property ensures that any process fragment eventually terminates (has the option to complete);
    - *Reachability of composition interfaces:* all fragment elements tagged with a composition interface can be reached from the start event. This property is required for the proper composition of business process fragments;
    - *Process fragments are deadlock-free:* implies that process fragments have no deadlocks, therefore they don't get stuck during their execution;
    - *Absence of dead tasks:* there are no tasks or activities in the process fragment that are never executed (no dead tasks);
    - *No infinite occurrence sequences:* all occurrence sequences (execution traces) of the process fragment are finite;
    - *Data type consistency of composition interfaces:* check that when two fragments are composed, the data types at their composition interfaces correspond. Ensures the good progression of data flow after a composition is performed and also the data compatibility of the composed fragments.
  - *Fragment specific:* certain properties cannot be verified in general and differ form one business process fragment to another and the specific context in which that process is used. We want to offer the product line engineer the possibility to define and verify such process specific properties. Therefore, we propose several *general property templates* which can be instantiated by the product line engineer for a specific purpose, for verifying a specific property of interest. These property templates allow answering certain specific questions about a process fragment, like:
    - Can a certain flow object be reached from the start event?
    - Is a certain flow object always executed?

  - Will a certain activity have a data object of a certain type during process execution?
  - If activity x is executed, will activity y also be executed?
  - If activity x has data type dx than activity y will have dy?
  - If activity x is executed, then activity y will never be executed

3.4.2 Business process fragment verification process

The entire process of verification of business process fragments is graphically depicted in Figure 3.5. As it can be noticed from the figure, the verification process consists of checking, in parallel, two types of properties: *structural* and *behavioural correctness*.

- **Verification of structural correctness:** this type of verification ensures that the business process fragments satisfy the *structural correctness* property described above. During this step of the process, the set of structural well-formedness rules are checked for the process fragments being verified. Both types of consistency rules are verified: the ones based on the OMG BPMN specification and the fragment specific ones. The goal of this type of verification is to point out structural flaws or inconsistencies in the business process fragments.

  As presented previously, there are two main ways for obtaining business process fragments: creating new fragments from scratch or reusing existing fragments from a fragment repository. For each case, the verification of structural correctness can be performed:

  - *For newly created fragments:* as the structural consistency rules are specified as OCL rules defined on the BPMN process fragment meta-model, the verification of structural correctness is performed automatically: any business process fragment created which is conform to the meta-model will verify all of the consistency rules. Therefore, our methodology guarantees that all business process fragments are *structurally correct by construction.*
  - *For reused fragments:* the verification of structural correctness can also be performed for fragments that are directly reused or reused with some modifications. The same set of consistency rules needs to be checked on the final fragment that will be used as an asset of the product line. These checks can be performed either manually by the product line engineer or by using an automatic verification tool.

- **Verification of behavioural correctness:** this type of verification ensures that the business process fragments satisfy the *behavioural correctness* property described above. There are several steps involved in the process:

  - *Mapping to HCPN:* behavioural properties cannot be directly verified on the BPMN process fragments. To perform this type of verification, the BPMN process fragments have to be mapped onto a formal language. The Hierarchical Coloured Petri Nets (HCPN) have been chosen as target formal language. We propose a model-to-model transformation that takes a BPMN process fragments as input and returns a hierarchical coloured petri net. The entire behavioural analysis is then performed onto the resulting petri net.

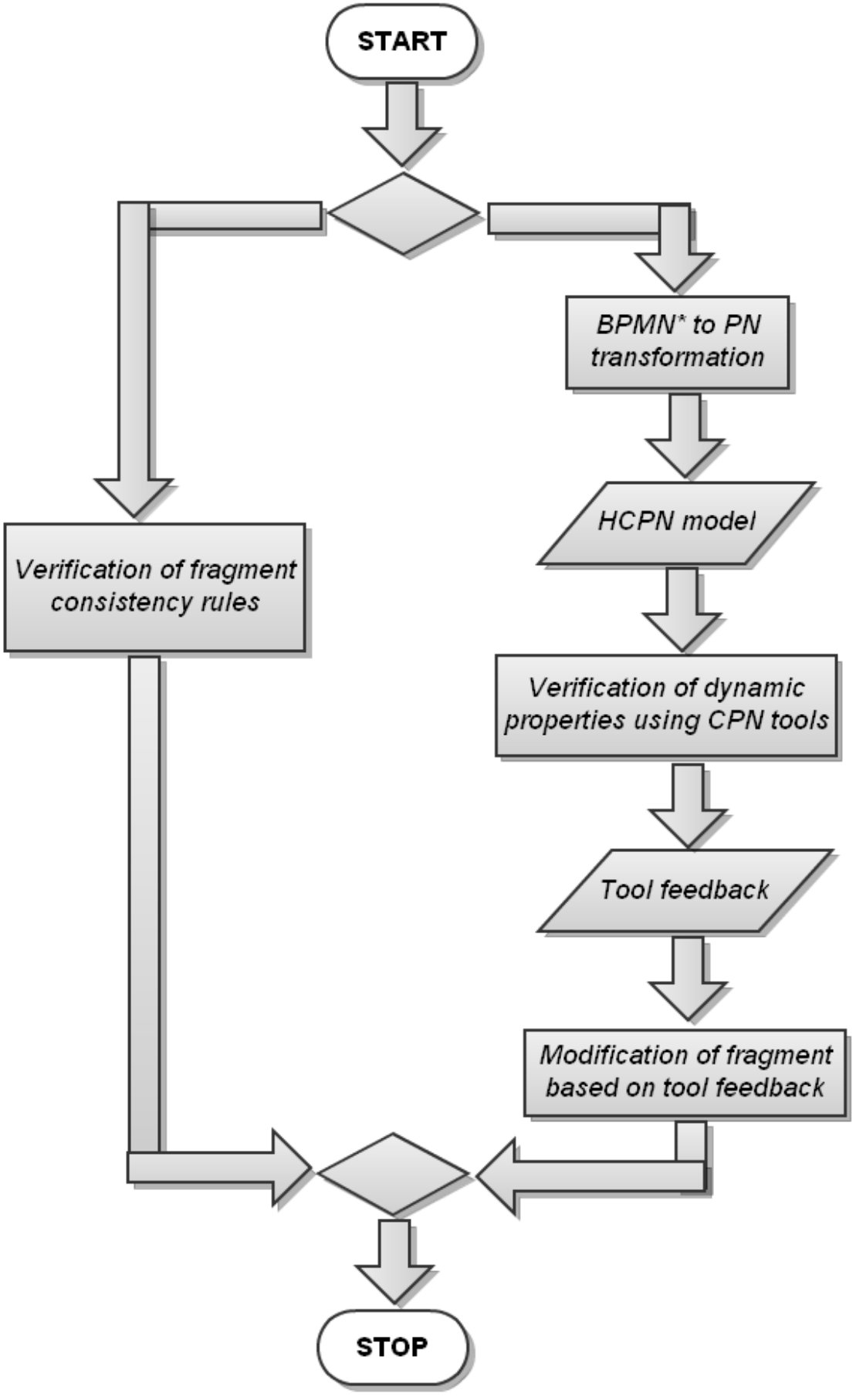


Fig. 3.5: Complete process of verification of business process fragments

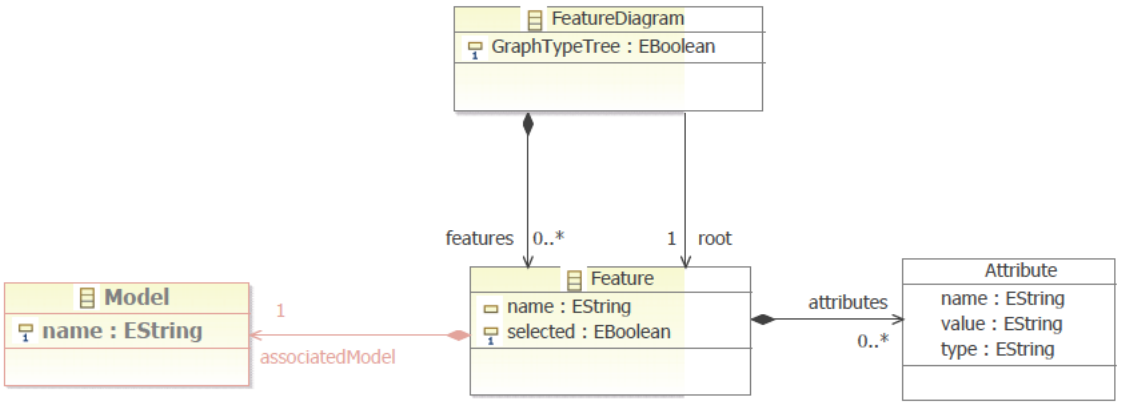


Fig. 3.6: Feature diagram meta-model: associating fragments to features

- *Verification of dynamic properties using CPN tools:* the goal of the mapping to HCPN is to allow access to the formal verification techniques and tools available at this level. Using CPN tools [RWL+03], both types of behavioural properties defined for business process fragments (general and fragment specific) can be verified.
- *Fragment modification based on tool feedback:* the analysis performed using CPN tools returns a result specified in terms of petri net concepts. Since the goal is to verify the business process fragment, this result needs to be interpreted in terms of BPMN concepts. Once this operation is achieved, the business process fragment under verification can be adapted and the identified errors solved.

3.5 Association of business process fragments to features

Variability modelling with feature models was performed in one of the initial steps of the methodology to capture the variability within a product line. Feature models abstract from concrete feature realizations. However, feature models usually do not exist alone, but are related to reusable assets describing the solution space. In order to build concrete products from a product line, features have to be realised using software artefacts shared across the product line. Throughout this step of the methodology, we aim at bridging the gap between feature models and solution models. Therefore, we define a mapping of features to model fragments specifying the concrete feature realisations.

While variability modelling resides in the problem space, the realisation of features is part of the solution space. To instantiate products from a product line, feature realisations in the solution space have to be included according to the presence of the features in a variant model that is an instance of a feature model. To support this transition from problem space to solution space in an automated way, a mapping from features to software artefacts that realise the features is needed. This mapping will also allow for the automatic derivation of a product instance based on a given variant configuration.

This phase of the methodology takes as input the feature diagram obtained during the *feature diagram construction* phase and the correct BPMN process fragments resulting from the *business process fragment verification* phase. As the business process fragments were

created based on the feature descriptions and their purpose is to be the concrete feature implementations, associating the fragments to the features is quite straightforward. The mapping is generic and connects features from the problem space with any type of product line core assets of the solution space. Any type of models can be associated with a feature. For our methodology, business process fragments are specific type of assets used.

This association is implemented in the feature diagram meta-model. A relevant excerpt of this meta-model is presented in Figure 3.6. In this figure, the *Model* meta-class is highlighted. A model is simply characterized by its unique name. There is no restriction imposed on the type that the model can have. Implicitly, any type of models can be associated to features. There is an *association relation* between the classes *Feature* and *Model* which models this connection. We also impose a *1-to-1 cardinality* to this relation, meaning that each feature has a unique model associated to it which represents its concrete implementation. From a technical point of view, each feature has a *container*. Any type of model element, model fragment or entire model can be added into such a container. Therefore, associating a business process fragment to a particular feature simply resumes to adding it to the feature's container.

The entire process of verification of business process fragments is graphically depicted in Figure 3.7. As it can be noticed from the figure, the process is an iterative one. First, one of the features from the feature diagram is selected. Then, the product line engineer needs to select the business process fragment that will be be associated to that feature. The selected business process fragment is then added into the container of the feature. One this operation is performed, the product line engineer checks if, in the feature diagram, there remain other features which have no process fragments associated. If this is the case, the entire previous procedure is repeated for a new feature. In case all the features have assigned implementations, the process is completed.

3.6 Configuration of the feature diagram

This step of the methodology is the first one that belongs to the Application Engineering phase. It consists of selecting the required features that will be part of a particular product that is derived. The actual feature selection process is based on user requirements and choices It highly, so this step of the methodology highly involves the end-user. We first explain what a *configuration of the feature diagram* means and then detail the steps involved in the configuration process.

3.6.1 What is a feature diagram configuration?

A feature model describes the configuration space of a product family. It represents a set of configurations, each being a set of features selected from a feature model according to its semantics. The product line engineer may specify a member of the product line by selecting the desired features from the feature model within the variability constraints defined by the model. These are instance of the feature diagram and consists of an actual choice of atomic features, matching the requirements imposed by the diagram. Such an instance corresponds to a product configuration of a system family.

Configuring a feature diagram is a technique intensively used in product line engineering. Several authors define this concept in different ways:

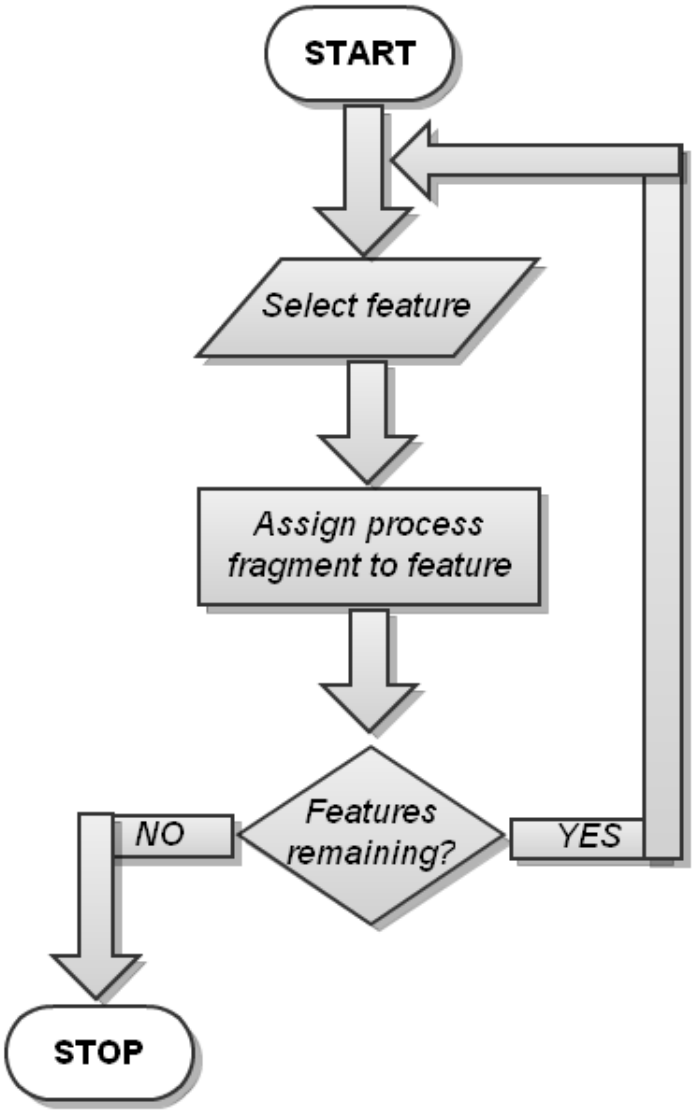


Fig. 3.7: Complete process of associating business process fragments to features from the FD

- In [Wik] a feature configuration is defined as *"a set of features which describes a member of an SPL: the member contains a feature if and only if the feature is in its configuration. A feature configuration is permitted by a feature model if and only if it does not violate constraints imposed by the model"*.
- Czarnecki et al. define feature configuration as *"the process of specifying a family member, whereas the specification is fulfilled by an stakeholder who selects the desired features from the feature model taking into account the variability constraints defined by the model as well as other constraints which could not be modelled as a feature diagram"* [CHE05b].
- In [BCTS06] configuration is presented as *"the process of deriving a concrete configuration conforming to a feature diagram by selecting and cloning features, and specifying attribute values"*.

Configuration is the process of deriving a configuration, selecting or removing features, from the feature diagram (while taking any constraint into account), in order to reduce the variability that the feature model is depicting. A configuration consists of the features that are selected according to the variability constraints defined by the feature diagram. The relationship between a feature diagram and a configuration is comparable to the one between a class and its instance in object-oriented programming.

The outcome of the configuration can be either a concrete configuration which uniquely identifies a product in the product line (because there is no point of variability) or a partial configuration which represents the variability of a subset of products in the product line; that is, a partial configuration is an specialisation because it yields another feature diagram.

Czarnecki et al. define in [CHE05b] the notion of *staged configuration* as the process where *"each stage takes a feature model and yields a specialised feature model, where the set of systems described by the specialised model is a subset of the systems described by the feature model to be specialised"*.

There are actually two possible configuration procedures, depending on how the process is carried out:

- *Configuration:* a complete and unique configuration is derived from the feature diagram;
- *Specialisation:* a sequence of partial configurations are derived, each one from the previous one, starting with the product line's feature diagram, and then a configuration is derived from the last staged configuration, which is a fully specialised feature diagram.

There are a few rules that apply when performing a feature diagram configuration, which Laguna et al. have identified in [LMRC11]:

- *Core selection:* since the root feature of any feature diagram is the smallest prospective configuration, any mandatory child feature of the root feature (and subsequently, any other mandatory child feature connected indirectly to the root feature through mandatory child features) also belongs to this "smallest" configuration;

- *Selection by inheritance:* any mandatory child feature of a selected feature should also be selected in the configuration;
- *Selection by parenting:* a non-mandatory child feature of a feature can only be selected (or included in the configuration) if it has at least one parent which is selected;
- *Decomposing selection:* when a feature is selected, the number of its child features which are to be selected should be not more than n but no less than m, for a cardinality of [m..n];
- *Selection by require constraint:* any feature which is required as a result of a selected feature for which there is a require constraint such that demands it should also be selected in the configuration;
- *Selection by mutex constraint:* any selected feature which is involved in a mutex constraint restricts the selection in the configuration of all other features taking part in the same mutex constraint.

3.6.2 Feature diagram configuration process

All the steps of the feature diagram configuration process are graphically depicted in Figure 3.8 and presented in the following:

- *Selection of features by the user:* this activity highly depends on the involvement of the end user. This activity takes as input the feature diagram and is based on the specific user requirements. Based on these particular requirements, the user will select a set of features, that constitute a particular feature diagram configuration and are the output of this step of the process.
- *Propagation of constraints, relations, dependencies:* this activity is based on the set of rules previously presented that apply when performing a feature diagram configuration. Several conditions restrict the possibilities the user may have when selecting different features during feature diagram configuration. The different variability relations that may exist for a feature group will impose certain restrictions to the possibilities that can be made. For example, in case of the XOR operator, once a feature is selected, the other ones are implicitly excluded from the current configuration, no matter if the user wants to select some of them. Similarly, when group cardinalities are present, the upper and lower limit need to be taken into account when making the feature selection. Cross-tree feature dependencies might also impose the presence of certain feature in the current configuration or completely exclude other ones. Therefore, all the rules and constraints of the feature diagram are propagated every time the user selects a feature for the configuration.
- *Business process fragment selection:* in the previous step of the methodology, the core assets of the product line, the business process fragments, were related to features from the feature diagram. These associations define for each feature a concrete implementation. When a particular feature diagram configuration is achieved, the business process fragments associated to the set of feature that pertain to the configuration are also selected. This is an automatic process. The result is a set of business process fragments from which the behavioural model of the derived product will be constructed.

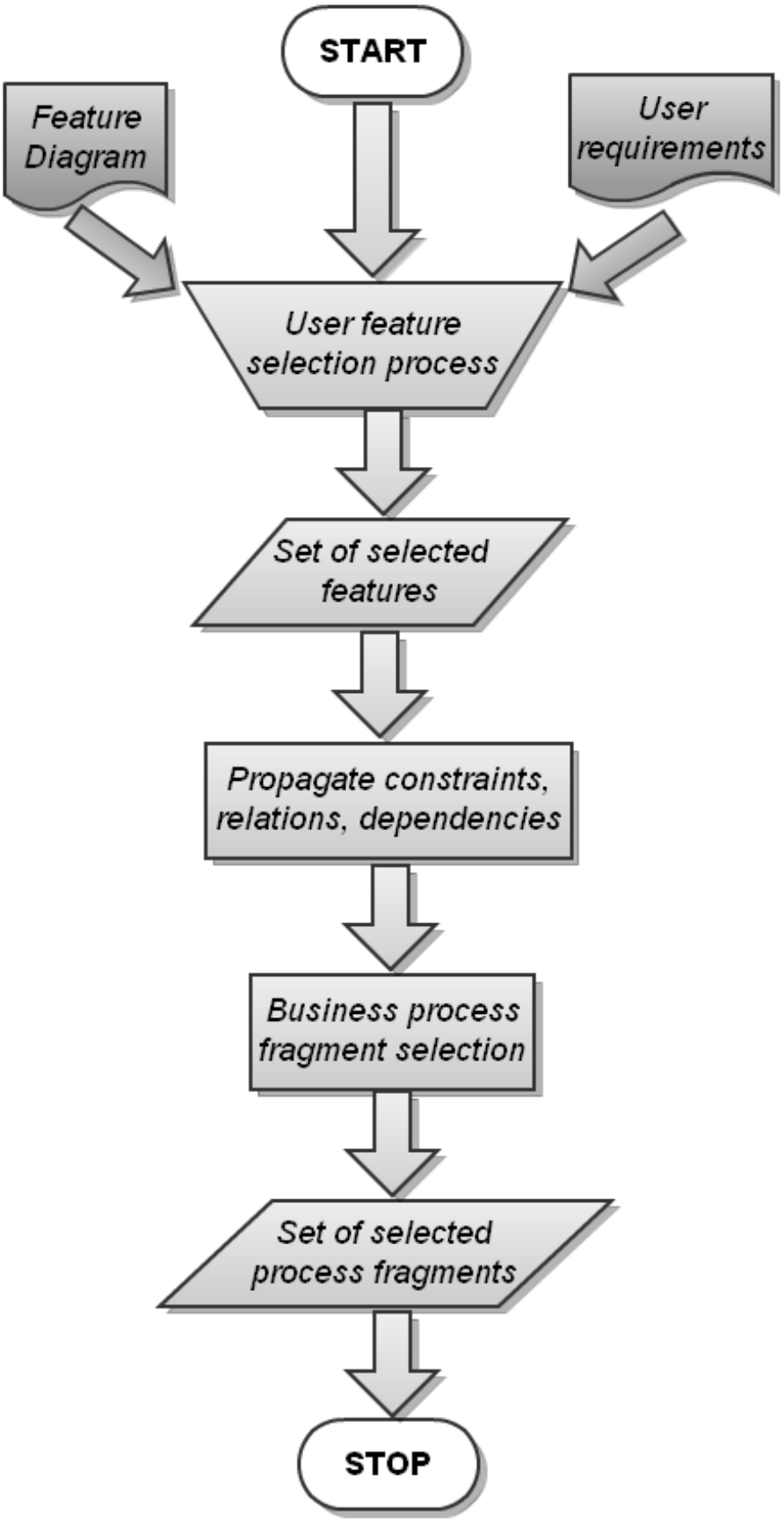


Fig. 3.8: Complete process of configuring the feature diagram

3.7 Product derivation specification

The last phase of the methodology is called *product derivation specification*. It takes as input the set of business process fragments resulting from the feature diagram configuration step and transforms them, through a compositional approach, into a proper business process that models the behaviour of the SPL product being derived. The compositional approach applied requires the introduction of a new concept, the *composition interface of a business process fragment*, and the definition of a set of *BPMN composition operators*. Both concepts are essential for the well-functioning of the composition process and will be discussed in more detail in this section.

3.7.1 Composition interfaces

A business process fragment is intended as a reusable granule for business process design and can allow for reuse of process logic. Process fragments represent incomplete process knowledge, which needs to be integrated with further process knowledge to become a complete process model. A process fragment represents the implementation of a single abstract activity or functionality.

The *business process fragment* concept is comparable to *reusable components* in software engineering. Component-based software engineering (CBSE) is a branch of software engineering that emphasizes the separation of concerns and is also a reuse-based approach to defining, implementing and composing loosely coupled independent components into systems. In CBSE, an individual software component can be a software package, a web service or a module that encapsulates a set of related functions.

Component-based software development is the process of assembling reusable software components in an application such that they interact to satisfy a predefined functionality. Each component will provide and require pre-specified services from other components, so the notion of *component interfaces* becomes an important concern. Interfaces are the mechanisms by which information is passed between two communicating components. Components offer interfaces to the outside world, by which it may be composed with other components [Cai00]. A software component communicates only through its interfaces.

In a similar manner, we consider imperative the introduction of interfaces for business process fragments. A process fragment interface explicitly defines the elements of a fragment where it can connect or be connected with other fragments. Interfaces are offered by one fragment in order to be used by other process fragments. An interface also defines the manner in which a business process fragment can be related with other fragments for compositional purposes. As process fragments represent incomplete process knowledge, the composition interface explicitly defines the places where they can be integrated with other process fragments to become a complete process model.

Composition interfaces allow business process fragments to be used as a black box for compositional purposes. For process fragments, composability is be achieved by using explicit interfaces for defining where the actual composition process will be performed. The presence of composition interfaces restricts the possible ways in which the actual composition of process fragments can be performed. It also creates compositional dependencies between process fragments, when several fragments need to be composed.

These interface can be seen as a signature of the business process fragments - the user

does not need to know about the inner workings of the fragment (implementation) in order to make use of it during composition. However, when a fragment needs to use another fragment in order to extend its functionality, it verifies its interface to determine their compositional compatibility. The goal of composition interfaces is therefore to enable and guide the composition process for business process fragments.

The concept of process fragment composition interface is introduced as part of the language we propose for modelling business process fragments and their composition. However, we start by first introducing the *composition tag* concept. A composition tag is a type of BPMN artifact, represented as a textual tag, that can be added on the flow objects of a business process fragment. It explicitly denotes the places where a business process fragment can be composed with other ones. Composition tags can be added on any type of flow object (activity, event or gateway). There are two distinct types of tags possible:

- *input tag*: the presence of this tag on a flow object of a business process fragment denotes that when this fragment is composed with another one, the composition process will be performed at this exact element, and that the second process fragment will be connected to the current one *exactly before the tagged flow element*. In other words, the process fragment is extended before the tagged flow object by composition;
- *output tag*: similarly, the presence of this tag on a flow object of a business process fragment denotes that when this fragment is composed with another one, the composition process will be performed at this exact element, and that the second process fragment will be connected to the current one *exactly after the tagged flow element*. In other words, the process fragment is extended after the tagged flow object by composition.

With the help of the composition tags just presented, we can introduce the notion of *composition interface*. It defines the exact places where a business process fragment can be composed with other fragments. We propose two types of interfaces:

- The *input composition interface* of a business process fragment is defined as the set of all its flow objects tagged with an input composition tag;
- The *output composition interface* of a business process fragment is defined as the set of all its flow objects tagged with an output composition tag;

Therefore, the composition interface of a business process fragment is the union of its input and output composition interfaces. A more detailed presentation of composition interfaces and of how they are used is available in Chapter 4.

3.7.2 Composition operators

The methodology presented throughout this thesis situation promotes the use of the separation of concerns (SOC) mechanisms as support for modelling complex business processes. In the SOC paradigm, concerns are defined separately and assembled into a final system using *composition operators* [SGS+04].

Business process composition [EN10] is regarded as a flexible mechanism capable to cope with the increasing complexity of business processes. Similar to component-based software development, the core idea is to create a complex business process by assembling simpler ones (fragments). Process composition reduces complexity by having smaller process components/fragments connected together by flexible mechanisms to realize a process that provides the same business support as the initial complex process. The complexity of building a business process is taken away from the business analyst and delegated to the actual composition.

Creating a business process by composition facilitates its understanding and its use. Moreover, it can be updated more easily, as the necessary changes are performed on smaller separate models. The maintainability of the business process is also enhanced. Another strong argument motivating the use of process composition is *process reuse* [MP08]. The desire to better manage processes and improve business efficiency has led to increased awareness and desire to reuse processes. Process reuse is a way to promote the efficiency and quality of process modelling. Fewer business processes are built from scratch, as many existing processes are used for the development of new ones, following a compositional approach.

The general approach when applying model composition is to provide *composition operators*. They are mechanisms that take two (or more) models as input and generate an output that is their composition. Most languages provide a fixed set of composition operators, with explicit notations, specific behaviour and defined semantics. In case a language does not provide a composition operator with the desired behaviour, different workarounds need to be used.

In recent years, the Business Process Model and Notation 2.0 has received increasing attention, becoming the standard for business processes modelling. BPMN is the basis for the business process modelling and composition that is proposed in this thesis and will be discussed in the next chapter. A thorough analysis of the BPMN specification document reveals that the standard does not address in any way nor does it provide support for business process composition. However, there are several possible workarounds. Conversations and choreographies, used to model interactions and message exchanges between participants, are a possible solution. Gateways, normally used to express control flow, can be used together with sub-processes, global tasks or call activities as a possible way to express process composition. Sub-processes are ow objects used as an abstraction mechanism in BPMN. They are used to hide or reveal additional levels of business process detail. Therefore, they can be used for hierarchical process decomposition. The use of sub-processes is a possible workaround for replacing some composition operators, like refinement. Nevertheless, complex compositions like choice or synchronization cannot be expressed using sub-processes.

All these workarounds are very limited in terms of possible results that can be obtained. Composition of BPMN models currently requires specific knowledge in advance and takes up a lot of time and effort. There are no composition operators available for BPMN. They are necessary to achieve the composition of BPMN processes. Implicitly, composition operators are mandatory for composing business process fragments. Therefore, another contribution of this thesis is to propose a set of composition operators created specifically for composing BPMN models and business process fragments.

To successfully apply a composition operator we must know where a business process can be connected with other processes. These are the places where the actual composition is

performed. The proposed composition operators are defined in close connection with the business process fragment composition interface previously introduced. The two concepts work together: the composition interfaces guide the composition process and define the exact places where the composition operators are applied; then, the composition operators specify the steps to be performed for the actual composition.

We define a *set of 9 binary composition operators*, which take two business process fragments as input and produce another business process fragment as output. The composition operators proposed are: *sequential*, *choice*, *arbitrary sequence*, *parallel*, *parallel with communication*, *refinement*, *synchronization*, *discriminator* and *insertion*. The operators are inspired from well known and well defined composition operators proposed for the Petri Net language. The execution semantics of BPMN it in terms of enabling and firing of elements, based on a token-game. The behaviour of a business process can be described by tracking the path(s) of the token through the process. The dynamic behaviour of Petri Nets is also defined in terms of firing of transitions which triggers the passing of a token through the net. As the execution semantics of both languages are defined in a similar manner, we consider that Petri nets might provide useful composition operators that can also be applied to business process fragments.

The composition operators are part of the language for the modelling and composition of business process fragments proposed in this thesis. They are discussed in detail in Chapter 4. The semantics of each operator is defined using a translational semantics towards an equivalent Petri net composition operator. These composition operators are used during the product derivation process, for composing business process fragments into the final product behaviour. However, they are valuable by themselves and can also be used independently from the SPL context, whenever we need to compose two business processed into a new one.

3.7.3 Product derivation specification process

All the steps involved in the product derivation specification process are graphically depicted in Figure 3.9 and detailed in the following.

- *Annotation of business process fragments with composition interfaces:* the first step of the process takes as input the set of business process fragments obtained after the feature diagram configuration. These fragments need to be compose together into a unique business process model that gives the behavioural description of the derived SPL product. We argued before that composition interfaces are an essential part of the composition process and therefore of the entire product derivation. The process fragments available at this step of the process have no composition interfaces defined on them. Therefore, during this step of the process, composition interfaces are defined on the entire set of business process fragments. We choose to define the composition interface of a business process fragment only at this late stage of the methodology due to the fact that presence of composition interfaces highly restricts the possibilities of composition for a fragment. We want to keep the process fragments as reusable as possible, so they are stored in the process library/repository without being annotated. We also consider that annotation with composition interfaces will highly differ from one product line to another and also between the different products of the same SPl. The annotation is performed iteratively for each fragment, until all

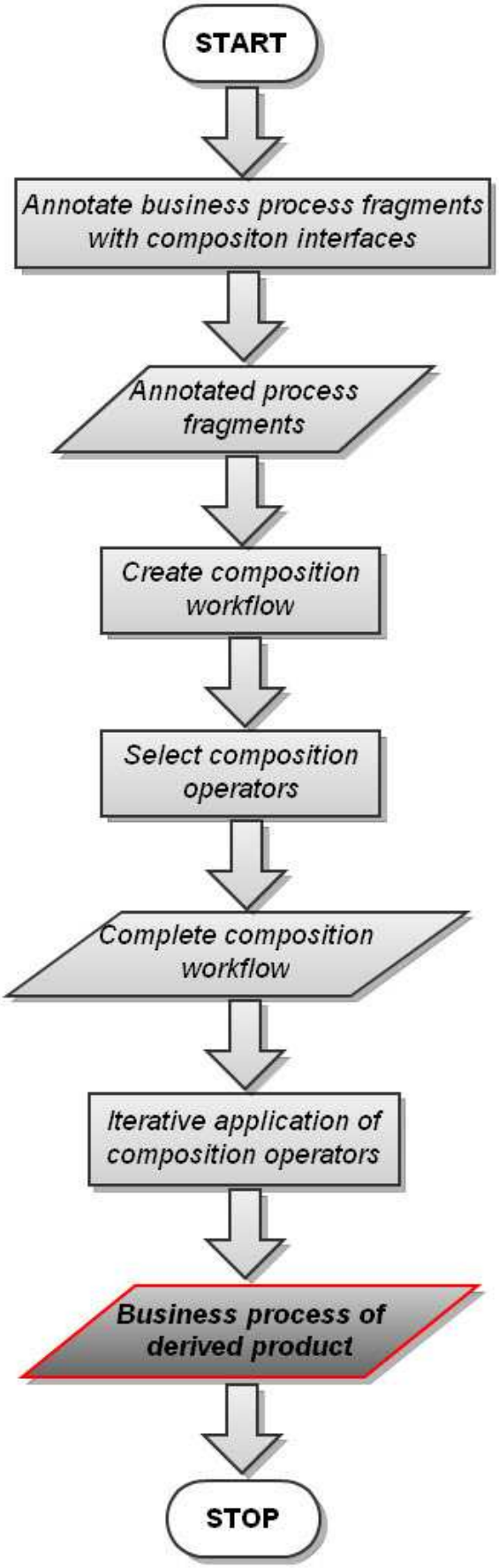


Fig. 3.9: Steps of the product derivation specification process

of them have been annotated. The result of this step is a set of annotated business process fragments which are ready for composition.

- *Creation of the composition workflow:* without any further guidance provided, several possible orders to compose the annotated business process fragments are possible. One could assume that all possible orders of composition lead to the same resulting product. It has been shown [IBK11] that this is actually not the case and that the order in which models are composed has a big influence on the result: different orders imply different derived products. To obtain a specific behaviour that characterizes the derived product, the process fragments thus need to be composed in a specific order. It is the product line engineer that defines this particular composition order, which is specific to each individual product that is derived from the SPL. We propose the use of a *workflow* for specifying this composition order. Thus, for a certain SPL, we will have as nanny composition workflows as the number of products we want to derive. The workflow is also created based on the specific composition interfaces of the business process fragments, which will highly restrict the possible orders.

  A composition workflow has several types of elements:

  - *Fragment place-holders:* for the composition workflow, business process fragments are seen as black boxes, we are not interested in their internal representations. Therefore, in order to reduce complexity, a composition workflow contains *fragment place-holders* instead of the actual business process fragments. A fragment place-holder references an actual business process fragment for further use;
  - *Operators:* the goal of the composition workflow is to specify the exact order in which process fragments are composed. It is essential to to be able to represent the different types of business process composition operators that can be applied. Therefore, another element of the composition workflow are the composition operators;
  - *Connectors:* we need to be able to represent the sequencing/flow of elements in the composition workflow. That is why we use simple *directed connectors*. A connector has a single source and a target and can connect a fragment place-holder to an operator or vice-versa.

  We try to keep the description of the composition workflow as simple and easy to use as possible, and therefore propose the minimum number of elements necessary.

- *Selection of applied composition operators:* once the composition workflow has been specified for the currently derived product and therefore the order in which the fragments should be composed defined, we need to specify the exact composition operators to be applied at each step. The product line engineer starts with the first two fragments from the composition workflow and decides which composition operator will be applied. Then, the hypothetical result of this composition needs to be composed with the next fragment present in the workflow. We therefore decide which operator should be used for this composition. In a similar manner, we traverse the entire workflow and decide on the exact composition operators that will be applied. At this stage, the composition workflow is complete and the actual composition process can take place.

- *Iterative application of composition operators:* the last step of the process involves applying the actual composition operators. The composition workflow is traversed and the following operations are performed:

  - The first two fragment place-holders from the workflow are selected, together with the operator that connects them. The place-holders reference the actual annotated business process fragments. Based on the composition interfaces of the two process fragments, the selected composition operator is applied.
  - We continue traversing the workflow and select the next fragment place-holder and the operator situated before it. The composition operator takes as input the process fragment resulting from the previous composition and the current process fragment.
  - Iteratively, we traverse the entire workflow and apply the same process for performing the actual compositions.

  Once the entire workflow has been traversed and processed, the result of the last composition is a business process that models the behaviour of the derived product and therefore the end result of our methodology.

The language support for creating the composition workflow is part of the business process fragment modelling and composition language that we propose in this thesis and will be discussed in more detail in the next chapter.

4. LANGUAGE FOR MODELLING AND COMPOSING BUSINESS PROCESS FRAGMENTS

Abstract

Business process fragments are the core assets used by our software product line methodology. The most common approach to obtain them is to create new business process fragments from scratch, as concrete implementations of the features from the feature diagram of the SPL. For this purpose, adequate language support is required. Throughout this chapter we propose a new domain specific language called CBPF created specifically for modelling composable business process fragments. We start by precisely defining in Section 4.1 what a business process fragment really is. We motivate the need for creating business process fragment and then detail several of their structural and behavioural characteristics. A model driven approach is then followed for creating and specifying the CBPF domain specific language. We start by defining the abstract syntax of the language in Section 4.2.We describe the high-level structure of the CBPF language by means of a meta-model representing in an abstract way the concepts and constructs of the modelling language, and providing the means to distinguish between valid and invalid models. The abstract syntax is described in an incremental manner: we fiest define the core part of the language; then, we enrich it with concepts related to the "composability" of business process fragments; finally, we further extend the language to support the modelling of product derivation specifications. We continue the language description in Section 4.3 by we proposing a unique graphical concrete syntax for the language. It is a crucial element of language design and we therefore treat it as a separate element within the language description. We conclude the chapter by defining the semantics of the CBPF language following a translational approach, by proposing a mapping of CBPF concepts onto the Hierarchical Coloured Petri Net (HCPN) formalism. The semantics can be seen as the abstract logical space in which models, written in our language, find their meaning. Semantics are as important as the structure of the language.

Business process fragments are the core assets used by our software product line methodology defined in Chapter 3, from which the models of the derived SPL products are created, following a compositional approach. The concept was introduced in Chapter 3, where we also presented the process followed for creating them. The most common approach is to create new process fragments from scratch, as concrete implementations of the features from the feature diagram of the SPL. For this purpose, adequate language support is required. Throughout this chapter, we propose a new domain specific language called *Composable*

Business Process fragments (CBPF) designed specifically for modelling such composable business process fragments. CBPF provides the necessary language support for several steps of our methodology. CBPF is based on the BPMN process modelling standard. It only uses a subset of the core BPMN elements, those proven to be the most commonly used when modelling business processes. Moreover, we introduce new concepts like *composition tags* and *fragment composition interface*, which facilitate the composition of business process fragments. Furthermore, we extend the language with a set of composition operators that enable the composition of business process fragments.

A model driven approach is followed for creating and specifying the *CBPF domain specific language*. The structure followed for the CBPF language definition is the following:

- *Defining the abstract syntax:* we describe the high-level structure of the CBPF language by means of a meta-model representing in an abstract way the concepts and constructs of the modelling language, and providing the means to distinguish between valid and invalid models. The meta-model of the CBPF language describes the vocabulary of concepts provided by the language, the relationships existing among those concepts, and how they may be combined to create valid models. We define first the core of the CBPF language and then extend it with composition interfaces and composition operators. Finally, we explain how this language can also support the creation of product derivation specifications.

- *Defining the concrete syntax:* we propose a unique graphical concrete syntax for the language. It is a crucial element of language design and we therefore treat it as a separate element within the language description. The proposed graphical syntax is inspired from the one of the BPMN language. We extend it with graphical representations associated to the newly introduced language concepts.

- *Defining the semantics:* semantics descriptions of software languages are intended for human comprehension. The semantics can therefore be seen as the abstract logical space in which models, written in our language, find their meaning. Semantics are as important as the structure of the language. We propose a translational semantics for the language. It is specified by mapping the current language onto another language that is formally well defined and understood, in our case hierarchical coloured Petri nets.

4.1 What is a composable business process fragment?

Business process fragments represent the core assets and the building blocks for our software product line methodology. They are created during the *business process fragment construction* step of the methodology 3.3, then the behavioural models of the derived products of the SPL are obtained by composing such fragments during the *product derivation specification* phase 3.7. Throughout this section we explain what a composable business process fragment is and which are its main characteristics.

It is important when designing or modelling processes to create them taking reuse into account right from the start. To bring more dynamics and flexibility into reuse in business process modelling, we need a more modular and granular way to define and describe reusable parts of business process models. A *business process fragment* is intended as a

reusable granule for business process design and can allow for reuse of process logic. This concept is comparable to reusable components in software engineering.

The concept of *process fragments* has already been used in the research literature. To be able to refer to different parts of a process model, several authors have defined process fragments as *connected parts of a process model*, where boundary nodes of a process fragment can be distinguished as fragment entries and fragment exits based on the directions of incident control flow edges. However, this type of definition of a process fragment (a connected sub-graph of a process graph) does not coincide with our view and is not our intention. We consider business process fragments as *self-contained connected process structures*, which are in most cases created from scratch in a bottom-up approach. Process fragments are designed to implement a set of requirements and model a single abstract functionality, and thus are not a sub-graph that can be extracted from a pre-existing process graph. Therefore, business process fragments can be considered on their own as independent units of reuse for business process modelling.

Another essential characteristic of business process fragments is their capacity to *represent incomplete process knowledge*, which needs to be integrated with further process knowledge/information in order to become a complete process model. This makes them incomplete building blocks, containing some local process knowledge that might be useful for more that one business process. This property is essential as business process fragments model a single high-level functionality, therefore the provided model only offers information about that functionality and might not be complete. Therefore, a business process fragment is not necessarily directly executable and it may be partially undefined.

By definition, business process fragment also have to be *composable* together into a business process and leave some room for adoptions where the exact business logic is not known at fragment design time. A process fragment represents the implementation of a single abstract activity or functionality. In order to describe the features of a product, these functionalities are meant to be combined. Therefore, business process fragments are conceived with the idea of future composition in mind and will have precisely defined elements inside the fragments denoting the places where they can connect with other fragments.

From a structural point of view, a business process fragment is a *self-contained block of process logic with strictly defined boundaries*. Form a semantic perspective, a business process fragment can be interpreted as a *detailed specification of a high level abstract task or functionality*. A process fragment is accepted as a unit of meaningful aggregation of process logic. They need to be coherent and make sense to a domain specialist. Each fragment forms a useful resource in its own right.

We require several characteristics from a business process fragment:

- a business process fragment may have *significantly relaxed completeness and consistency criteria* compared to an executable business process. This property is in direct relation with the fact that business process fragments model partial or incomplete knowledge and are meant to be integrated with other fragments. Implicitly, the completeness and consistency criteria that apply to regular business processes need to be adapted for fragments;
- we require that a business process fragment be *structurally correct*. The notion of structural correctness is discussed in-depth in Chapter 5. It is an important property, ensured through the definition of several well-formedness and consistency rules defined directly on the business process fragment meta-model;

- business process fragments are conceived as *units of process reuse*. One of the key aspect of business process fragments is their ability to enable process reuse across different product lines and within different companies. They are created once but may be used several times directly or with possible modifications or adaptations for different products of one SPL or for different product lines all-together.
- business process fragments are meant to be *composable*. Therefore, a process fragment will contain specific areas where it can connect or be connected with other process fragments through composition;
- lastly, a business process fragment has to consist of at least one start event (entry point) and one end event (exit point). Moreover, as a process fragment models an abstract functionality, it is required to contain at least one activity.

Business process fragments may be used by different stakeholders, each with its own point of view and specific interest in using the fragment. To accommodate such different possible perspectives, we define *three possible fragment logical viewpoints*:

- *Fragment viewpoint*: it defines the actual or intended behaviour of the process fragment. This view corresponds to the actual workflow structure of the process, defining the flow of activities. It is the straight-forward way of interpreting a business process fragment. The product line engineer that creates the fragment is the one that is the most interested in this low-level, high-detail perspective.
- *Composition viewpoint*: addresses the fact that business process fragments area meant to be composed with other fragments. This viewpoint identifies the *composition interface* of a business process fragment. It specifies the exact places in the fragment where it can be composed with other ones. This view abstracts from the inner representation of the fragments and only focuses on the composition aspect.
- *SPL viewpoint*: defines the behaviour of the fragment as a "black-box", seen from the outside. Process fragments are meant to be concrete implementations of an abstract functionality. Therefore, this viewpoint abstracts from the actual implementation and process structure and focuses only on the functionality (feature) that the business process fragment is meant to implement. It is the most high-level way in which one can look at a business process fragment.

The use of business process fragments comes with certain advantages, which resemble those of code reuse in traditional programming:

- the same logic does not need to be specified over and over again and can be highly reused in different projects;
- the quality of process design is highly improved, and can be better assured when the process fragments used in the process have an efficient design;
- in case a better fragment is available for a particular task, it can simply replace the less efficient version stored in the repository/library;
- over time, the quality of the process logic that is reused increases with this approach.

Now that the concept of *composable business process fragments* has been defined, we require the appropriate language support to allow us to model such business process fragments. We therefore propose the CBPF domain specific language for modelling and composing business process fragments. The language is constructed in an incremental manner. We start by creating a language that will simply allow the modelling of *business process fragments*. We then add new concepts that allow to make those business process fragments *composable*. Moreover, as the obtained business process fragments need to be composed in our methodology, we extend the language with a set of composition operators that will enable the composition of the process fragments. Finally, during the product derivation specification step of the methodology, we need to specify the workflow that defines the composition of the fragments. Therefore, we also extend the language with the necessary concepts that allow the creation of such composition workflows. Throughout the following sections, we present this language following a model driven engineering approach, by defining the *abstract syntax*, *concrete syntax* and *semantics* of the language.

4.2 Abstract syntax

The *abstract syntax* describes the high-level structure of the CBPF language elements and their relation. Following a model-driven engineering approach, the abstract syntax of the language is defined by means of a meta-model representing in an abstract (and visual) way the concepts and constructs of the modelling language. It also provides the means (constraints) used to distinguish between valid and invalid models.

The meta-model of the language describes the vocabulary of concepts provided by our language, the relationships existing among those concepts, and how they may be combined to create models. We employ the use of a meta-model based abstract syntax definition as it has the great advantage of being suitable to derive from the same meta-model (through mappings or projections) different alternative concrete notations (textual or graphical or both) for various scopes like graphical rendering, model interchange, standard encoding in programming languages, while still maintaining the same semantics. Therefore, a meta-model could be intended as a standard representation of the language notation.

4.2.1 Relation with BPMN standard

The CBPF language that we propose in this chapter is inspired from the Business Process Modeling Notation (BPMN), an increasingly important standard for process modelling and has enjoyed high levels of attention and uptake in practice. Presented in Chapter 2, BPMN is a rich language that allows to define a multitude of business scenarios, ranging from internal process choreographies to inter-organizational process orchestrations, service interactions and workflow exceptions.

BPMN offers a wide range of modelling constructs, significantly more than other popular languages. Version 1.2 of BPMN consists of 52 distinct graphical elements: 41 flow objects, 6 connecting objects, 2 grouping objects and 3 artefacts. That implies a lot of vocabulary to learn and understand, given that each graphical element has a specific meaning and rules associated to it. Even the core BPMN element set contains 11 elements. The complexity of the BPMN language increases even more for version 2.0, and includes almost 100 elements. However, not all of them are equally important in practice as business analysts frequently use arbitrary subsets of BPMN.

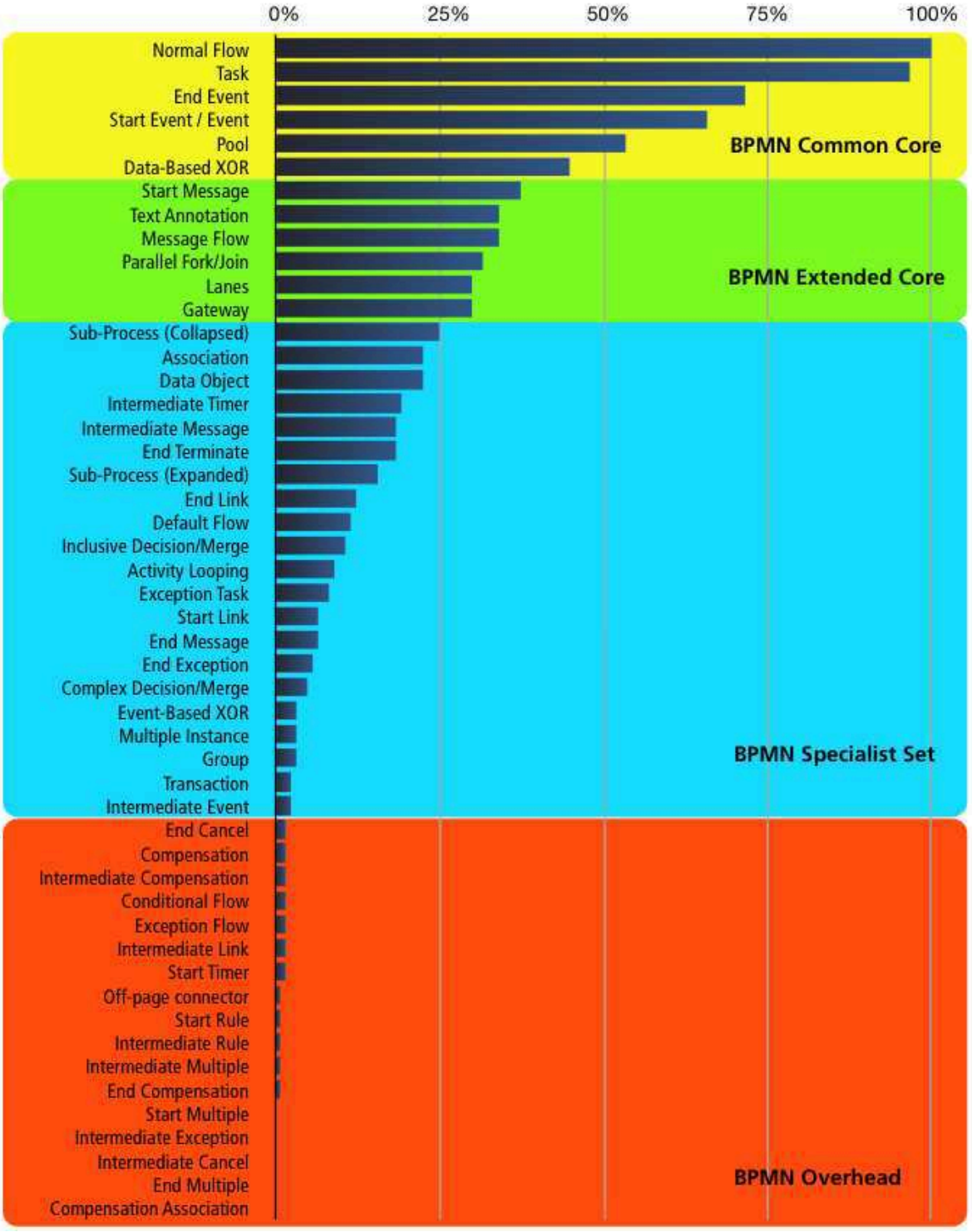


Fig. 4.1: Frequency distribution of BPMN construct usage [Rec10]

It should also be emphasized that one of the key goals of BPMN is to create a simple mechanism for creating business process models, and that the notation be simple and adoptable by business analysts. At the same time, BPMN has to be able to handle the complexity inherent to business processes. The approach taken to handle these two conflicting requirements was to organize the graphical aspects of the notation into specific categories. First, there is the list of core elements that will support the requirement of a simple notation. Most business processes will be modelled adequately with these elements. Second, there is the entire list of elements, including the core elements, which will help support requirement of a powerful notation to handle more advanced modelling situations.

Taking these aspects into consideration, we want to propose a language that is easy to use and understand by most people and which is not cumbersome to be learnt. We therefore ask ourselves the question: which of the BPMN elements are most used in practice and how frequently? The studies presented in [zMR08, RIRG06] provide valuable answers to this question. Figure 4.1 shows the frequency distribution of the individual BPMN constructs, separated by the three sample sets and ranked by overall frequency. It can be noticed that only four constructs being common to more than 50 % of the diagrams: Sequence Flow, Task, End Event and Start Event. All these constructs all belong to the BPMN core set. The figure also shows that every model contained the Sequence Flow construct, and the basic Task construct. The other BPMN constructs were unevenly distributed.

Based on these observations, the language we propose for modelling *composable business process fragments* (CBPF) will only contain a subset of the most used elements found in the BPMN standard. We consider that the selection of elements we have made allows to completely and correctly represent behavioural models that describe the business and operational step-by-step work-flows of activities. We remove the elements we consider not necessary for representing such types of models, in order to keep the language simple, clear and concise. The meta-model that defines the abstract syntax of the language is presented in the following.

4.2.2 Language meta-model

The abstract syntax of the language is defined by means of a meta-model representing the concepts and constructs of the modelling language. The meta-model describes the vocabulary of concepts provided by our language, the relationships existing among those concepts and how they may be combined to create models. We start the presentation of the CBPF abstract syntax by first defining all the necessary concepts for modelling business process fragments and specify the existing relations between these concept.

High-level language structure

A major goal of our language is to propose a notation that is simple and easily adoptable. The approach taken to handle these requirements is to *organize the elements of the notation into specific categories*. This results in a small set of notational categories, which enable the reader of a diagram to easily recognize the basic types of elements and understand the diagram. Within these basic categories, additional variation and more detailed and specialized information can be added to support the modelling of more complex diagrams, without drastically changing the basic look and feel of the diagram.

Therefore, we propose to structure the abstract syntax of the CBPF language into *five basic categories of elements*:

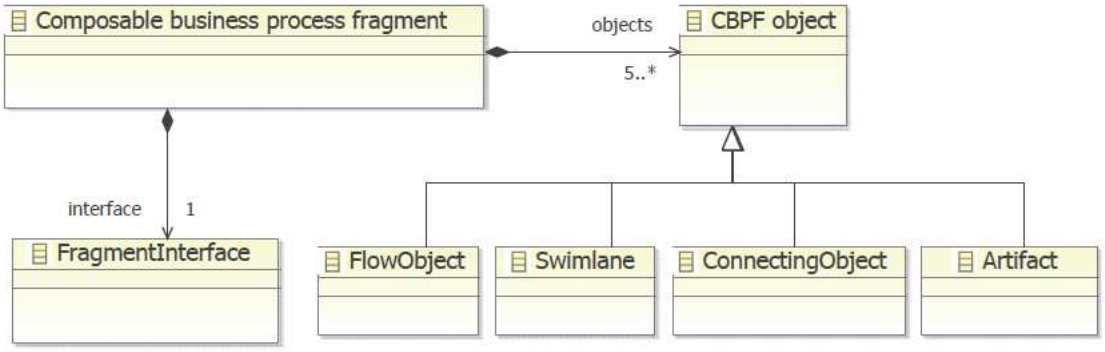


Fig. 4.2: Core structure of the business process fragment modelling language

- *Flow Objects:* are the main graphical elements, used to define the behaviour of a business process fragment;
- *Connecting Objects:* define the possible ways for connecting different flow objects;
- *Swimlanes:* define a visual way of grouping and organizing the primary modelling elements of a business process fragment;
- *Artifacts:* allow developers to bring more detailed information into the model/diagram, making it more readable;
- *Composition interface:* is a *newly introduced concept* that specifies the exact places where a business process fragment can be composed with other fragments.

The core structure of the language is graphically depicted in Figure 4.2. The root meta-class is *Composable business process fragment*, which denotes the entire language. It contains several *CBPF objects*, which denote the basic categories of elements proposed by the language. The meta-classes: *FlowObject*, *ConnectingObject*, *Swimlane* and *Artifact* represent exactly the element groups defined before. It can also be noticed that every business process fragments has a *unique composition interface*, denoted by the *FragmentInterface* meta-class.

Language concepts similar with BPMN

In the following, we discuss in more detail the different categories of elements and the language concepts contained by each one. The part of the language that corresponds to elements that also appear in the BPMN standard, which have been described in detail in section 2.3.3, will not be discussed in-depth.

Flow objects are the core concept in a business process fragment, defining its behaviour. There are three different types of flow objects:

- *Events:* are something that happens during the course of a business process. In our meta-model, the *Event* meta-class inherits from and is a subclass of the *FlowObject* meta-class. The main attribute of an event is its *type*. We define three types of events:

  - *Start:* indicates where a particular business process fragment starts and are mandatory within a business process fragment;
  - *Intermediate:* denotes that something happens inside the flow of the process fragment;
  - *End:* indicates where a process finishes and are mandatory within a business process fragment. We allow the existence of *multiple end events* in a business process fragment (there may exist several ways in which a business process fragment terminates).

  Every event has a *trigger*, which defines the cause for that event. The BPMN standard proposes a set of 10 types of event triggers. For modelling business process fragments and using them to represent product behaviour in SPL, we only consider necessary to propose the following triggers in our language:

  - *Message:* has a string attribute called *message* that specifies the exact text to be transmitted during the message exchange;
  - *Timer:* has a *TimeDate* attribute defined for reflecting a specific time-date or a specific time cycle;
  - *Plain:* most generic type of trigger and can be applied to any type of event;
  - *Error:* has a unique *ErrorCode* string attribute that identifies the specific type of error detected. Also, intermediate error events are usually attached to an activity.

- *Activities:* are the main elements of a business process fragment. We allow for *looping activities* in CBPF. If the *isLooping* boolean attribute is True, the concerned activity will be executed several times. In this case, the looping activity will have a boolean expression (*loopCondition*) that is evaluated after each cycle of the loop. There are two distinct types of activities: *task* and *sub-process*.

  - *Task:* are the atomic units of behaviour in a business process fragment. Each task has an *operation* which defines the specific activity that a task performs. We impose the condition that each task has a *single input and a single output sequence flow*;
  - *Sub-process:* is characterized by the *type* attribute, which can be either *collapsed* or *expanded*.

- *Gateways:* have a *type* attribute which determines if the behaviour of the gateway is *splitting* or *merging*. There are four sub-types of gateways possible:

  - *Exclusive:* creates a forking of paths for a business process process fragment. However, only one of the paths can be taken. The choice of which path to follow is made based on a *decision*.
  - *Inclusive:* represents a branching point where alternatives sequence flows may be followed. All sequence flows with a True evaluation will be traversed. Since each path is independent, all combinations of the paths may be taken, from zero to all;
  - *Parallel:* provide a mechanism to synchronize or to create parallel sequence flows within a business process fragment.

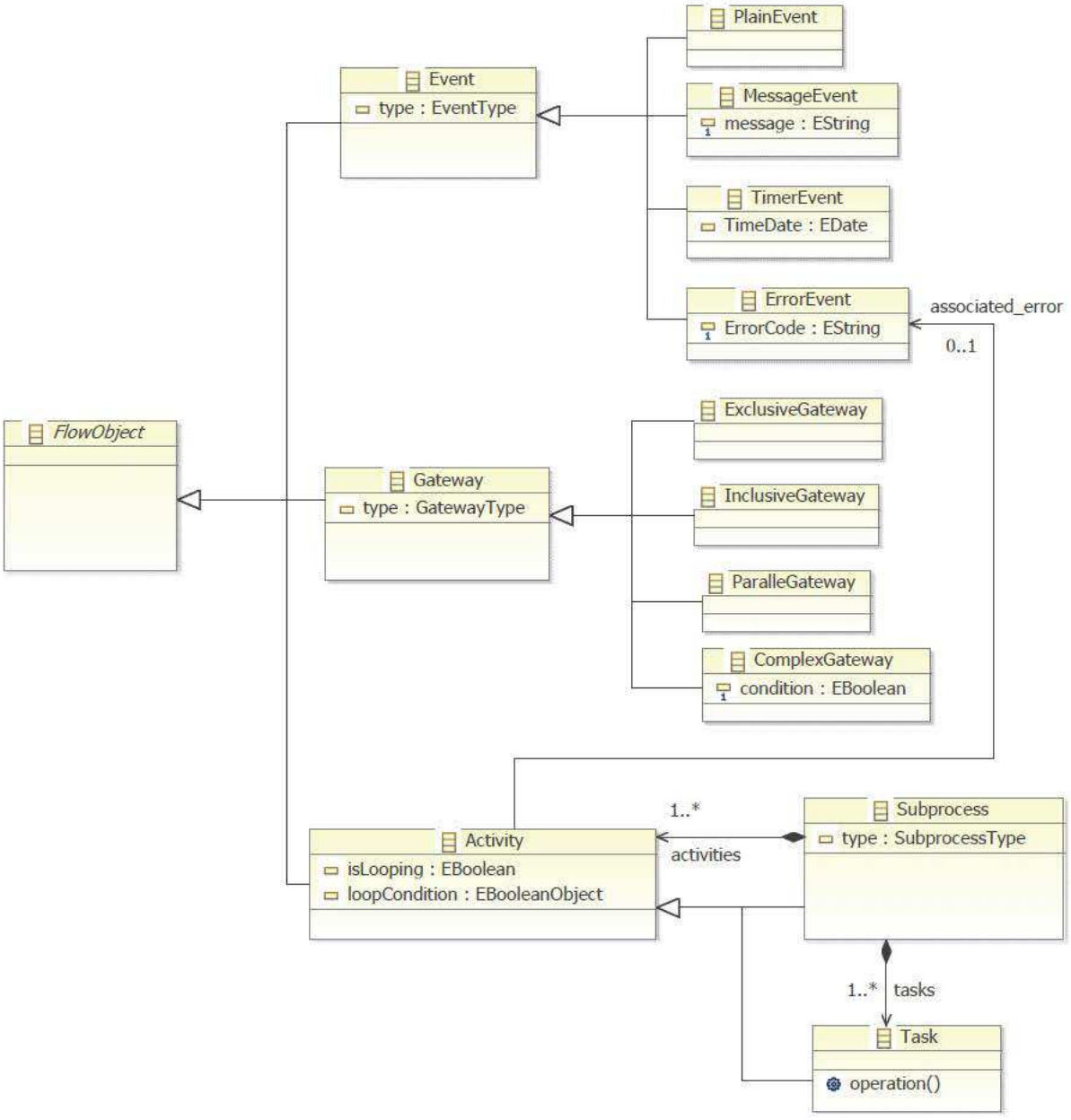


Fig. 4.3: Excerpt of composable business process fragment meta-model: flow objects

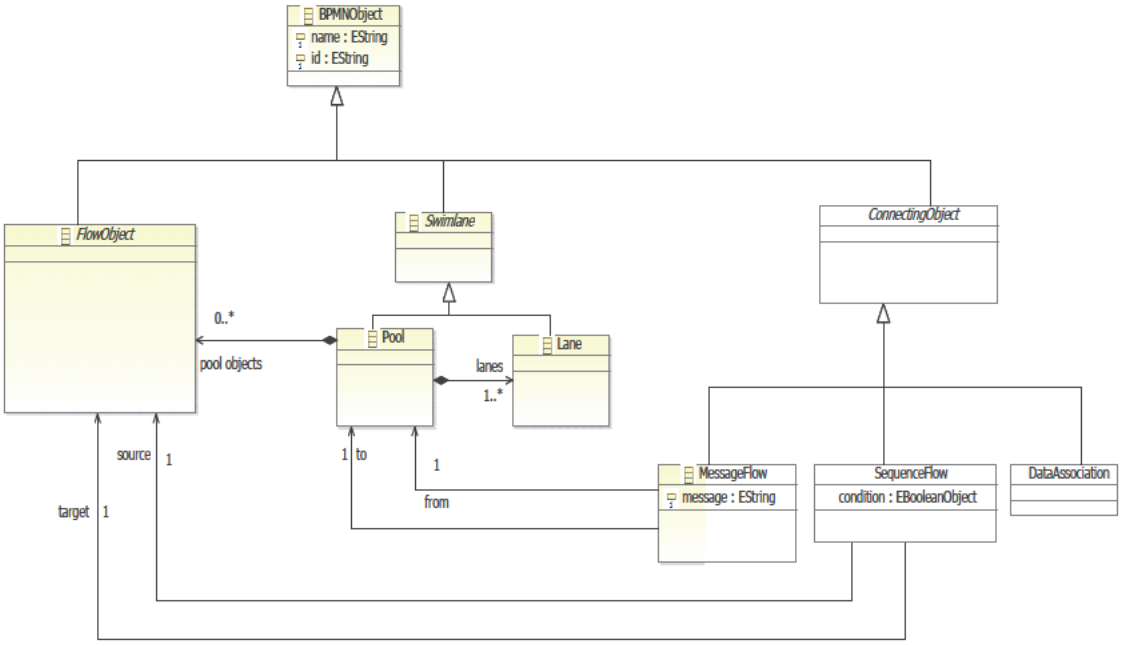


Fig. 4.4: Excerpt of language meta-model: swimlanes and connecting objects

– *Complex:* contains a boolean attribute called *condition* that specifies an expression that determines which of the sequence flows will be chosen for the process fragment to continue.

The part of the language meta-model that defines the flow objects is depicted in Figure 4.3.

Swimlanes are used to help partition and organize the activities of a business process fragment. Their goal is to represent participants of a business process and their collaboration. There are two kinds of swimlanes: *Pools* and *Lanes*.

- *Pools:* represent participants in a business process fragment. A pool is in general a container and *regroups several flow objects*, representing the work that the pool needs to perform under the process fragment being modelled.
- *Lanes:* are sub-partition of pools. Lanes are mainly used to organize and categorize the activities within a pool.

Connecting objects are used for connecting together the flow objects in a diagram. There are three possible types:

- *Sequence flow:* shows the order in which flow objects are performed in a business process fragment. A sequence flow has only *one source* and only *one target*, which must both be flow objects. A sequence flow has a boolean attribute called *condition*. This means that the condition expression must be evaluated before traversing the flow. Such conditions are usually associated with exclusive gateways, but may also be used with regular activities.

- *Message flow:* is used to show the flow of messages between two separate business process fragment participants. Message flow has a single *source* and a single *target*. It is mandatory that message flow connects two flow objects that belong to different pools.

The part of the language meta-model that defines *swimlanes* and *connecting objects* is depicted in Figure 4.4.

Artifacts allow developers to bring more information into the business process fragment. In this way the model becomes more readable. From the original BPMN specification, we do not use *groups* or *text annotations*, which we consider not necessary for the goals of our language.

Newly introduced concepts

Besides the previously presented elements inspired from the BPMN standard, we also propose a set of new concepts, specific to business process fragments. These new concepts are introduced in the language either as types of artifacts or as types of connecting objects. These concepts serve mainly two purposes:

- *Data objects*, *data specifications* and *data associations* are used for representing data and data flow in a business process fragment;
- *Composition tags* and *composition interfaces* facilitate the composition of business process fragments.

These concepts are presented in detail in the following:

- *Data association:* this is a new concept, that does not appear in the BPMN standard. Data associations are a specific type of connecting object. They are used to associate data objects with flow objects. Associations are needed to show the data inputs and outputs of activities. A data association relation has as single source a task of the business process fragment, and as single target a data object of the business process fragment. Associations are also used to model the data flow of a business process fragment.
- *Data objects:* we propose this concept to allow the modelling of data and data flow in a business process fragment. Very often, when executing a business process fragment, there may be data produced, either during or after the end of the process. A traditional requirement of process modelling is to be able to model the items (physical or information items) that are created, manipulated, and used during the execution of a process fragment. Thus, we propose data objects as a mechanism to show how data is required or produced by the activities of a business process fragment. Data objects are introduced in the language as specific types of artifacts. The presence of this concept allows to represent the data flow of a business process fragment. Several data analysis, ranging from simple to complex one, can be thus performed on a business process fragment. However, data flow analysis and processing for business process fragments is not addressed during this thesis, but is part of the future work. Thus, we only propose in this section a way to represent data for business process fragments but don't go into further details regarding this topic.

Activities often required data in order to execute. In addition, they may produce data during or as a result of execution. Therefore, we propose two types of data objects:

- *Data input:* is a declaration that a particular kind of data will be used as input to a task in order for that task to execute. There may be multiple data inputs associated with a task;
- *Data output:* is a declaration that a particular kind of data may be produced as output of the execution of a task. As before, there may be multiple data outputs associated with a task.

Data objects need to be associated with flow objects. The *data association* relation, introduced before, is used to make the connection between a flow object and its associated data objects. This relation is also depicted in the language meta-model in Figure 4.5, where the *Association* meta-class has two reference relations: one *source*, which is a task of the process fragment, and one *target*, which is a data object. This means that the behaviour of the process fragment can be modelled without data objects for modellers who want to reduce complexity and abstract from any data representations. The same process fragment can be modelled with data objects for modellers who want to include more information without changing the basic behaviour and flow of the process fragment.

Several other elements still need to be defined in order to complete the representation of data for a business process fragment. We introduce the notion of *data specification* as the general representation of the data that a task requires or produces. It can be observed in Figure 4.5 that the *Task* meta-class may contain a *DataSpecification*. The cardinality of this containment relation denotes the optionality of the data specification. This means that modellers are not obliged to represent data in their diagrams. In cas a data specification exists, it will be unique. Suc a data specification regroups all the data dependencies of a task. It contains one *InputSet* of data that is processed by the task, and one *OutputSet* of data that is produced by that task. The goal of the input set is to regroup all the data input objects that are required by a task. Similarly, the output set regroups all the data output objects that are attached to that task. The overall goal is to represent the fact that a task may have multiple input and output data objects associated with it.

Finally, every data object will have a unique *type.DataType* meta-class denotes the specific type of data contained by each data object. We propose three elementary data-types:

- *IntObject:* represents integer data objects;
- *StringObject:* represents string data objects;
- *BoolObject:* represents boolean data objects.

As presented in Chapter 3, during the application engineering phase of our SPL methodology, business process fragments need to be composed in order to obtain the behavioural models of the SPL products that we are deriving. To successfully realize these compositions we must know where a business process fragment can be connected with other ones. These are the exact places where the actual composition is performed.

In its current state, the CBPF language does not provide any support for specifying the exact places in a business process where the actual composition is performed. We therefore need to extend the language with new concepts that will enable the modelling of "composable" business process fragments. To facilitate the composition of business process fragments, we introduce two new concepts: *composition tag* and *composition interface*. They serve to render business process fragments *"composable"*.

- *Composition tags:* this is a newly introduced concept which we propose for business process fragments. By using such composition tags a business process fragment can easily and directly be composed with other fragments. A composition tag is simply a *text annotation* under the form of a *stereotype* that can appear on different elements of a business process fragment. We impose the constraint that composition tags can only be added on the flow objects of a business process fragment. This can also be seen in Figure 4.5, where the meta-class *FlowObject* has a containment relation with the *Composition tag* meta-class. The cardinality of this relation, which is set to 0..1, implies that at most one composition tag can appear on a flow object. Moreover, this also implies that there may exist flow objects with no associated composition tag.

  Composition tags are mainly used to guide the composition of business process fragments, specifically during the composition process itself. A composition tag identifies an exact place in a business process fragment where that fragment can be composed with other ones. This means that during the composition process, the process fragment elements tagged with composition interfaces will be directly involved. The specific operations particular to each composition operator will concern those tagged elements. The explicit identification of the composition areas with composition interfaces facilitates and guides the actual composition process. The specific manner in which composition interfaces are used in the composition process will be further detailed later on in this chapter, when the set of newly proposed process fragment composition operators are introduced.

  We propose and distinguish between two different types of composition tags:

  - *Input tags:* the presence of this tag on a flow object of a business process fragment identifies this element as the exact location where the actual composition with another fragment will be performed. It also specifies how, in a binary composition, the second business process fragment will be connected to the current one: the second operand is connected (added) *exactly before the tagged flow object*. Thus, an *input tag* requires an extension towards the top of the current process fragment. In other words, the process fragment is extended, by composition, before the tagged flow object;
  - *Output tags:* similarly, the presence of this tag on a flow object of a business process fragment identifies this element as the exact location where the actual composition with another fragment will be performed. It also specifies how, in a binary composition, the second business process fragment will be connected to the current one: the second operand is connected (added) *exactly after the tagged flow object*. Thus, an *output tag* requires an extension towards the bottom of the current process fragment. In other words, the process fragment is extended, by composition, after the tagged flow object;

  The part of the language meta-model that defines the *newly introduced concepts*

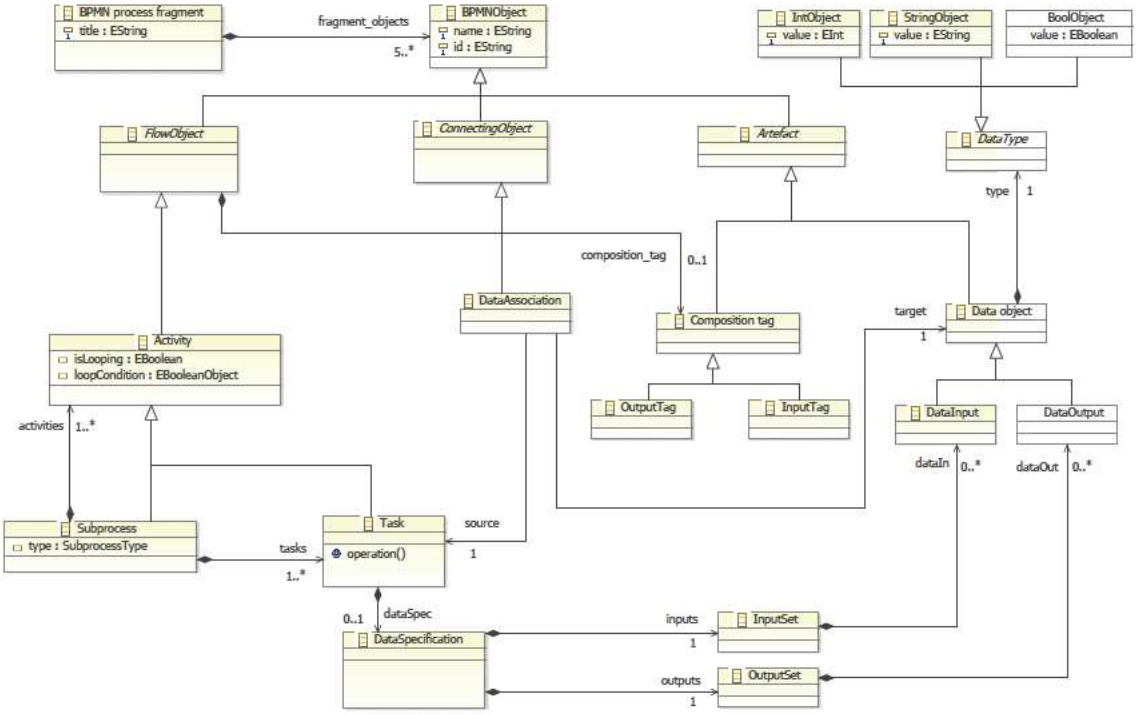


Fig. 4.5: Excerpt of language meta-model: newly introduced concepts

discussed above and how they are integrated in the language is depicted in Figure 4.5.

- *Composition interface:* business process fragments are intended to be reusable granules for business process design and should allow for reuse of process logic. They are comparable to reusable components in software engineering. Each software component will provide and require predefined services from other components, so the notion of *component interface* becomes an important concern. Interfaces are the mechanisms by which information is passed between two communicating components. Components offer interfaces to the outside world, by which they may be composed with other components.

  Based on the same principles, we propose the new notion of *composition interface* for a business process fragment. This concept allows business process fragments to become *composable*. Business process fragments represent incomplete process knowledge, which needs to be integrated with further process knowledge to become a complete process model. Therefore, in order to create complete business process models, business process fragments need to be composed together. The composition interface facilitates this activity. A composition interface explicitly defines the elements of a business process fragment where it can connect or be connected with other fragments. Interfaces are offered by one fragment in order to be used by other process fragments. An interface also defines the manner in which a business process fragment can be related with other fragments for compositional purposes. As process fragments represent incomplete process knowledge, the composition interface explicitly defines the places where they can be integrated with other process fragments to become a complete process model.

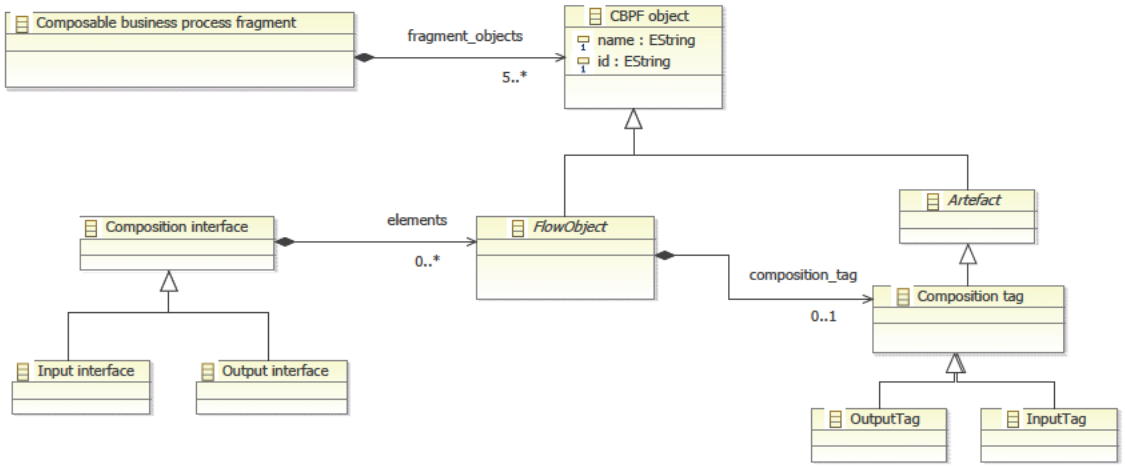


Fig. 4.6: Excerpt of language meta-model: composition interface

Composition interfaces allow business process fragments to be used as a black box for compositional purposes. For process fragments, composability is be achieved by using explicit interfaces for defining where the actual composition process will be performed. The presence of composition interfaces restricts the possible ways in which the actual composition of process fragments can be performed. It also creates compositional dependencies between process fragments, when several fragments need to be composed.

These interface can also be seen as a signature of the business process fragments - the user does not need to know about the inner workings of the fragment in order to make use of it during composition. However, when a fragment needs to use another fragment in order to extend its functionality, it verifies its interface to determine their compositional compatibility. The goal of composition interfaces is therefore to enable and guide the composition process for business process fragments.

This concept is an important part of our language and is integrated in the language meta-model. This can be seen from Figure 4.6, where a business process fragment has a unique *Composition interface*. This interface might be empty, meaning that the fragment does not define any places for composition. If not empty, a composition interface contains several flow objects. There is a close relation between the notion of *composition tag*, defined previously as part of the artifacts of the language, and the concept of composition interface. The composition interface is defined as a union of flow objects that have composition tags associated with them. We propose two types of composition interfaces, so there are two sub-classes that inherit from the *Composition interface* meta-class:

- *Input interface:* is defined as the set of all its flow objects tagged with an input composition tag. Implicitly, it defines all the places of a business process fragment where, during the composition process, the actual composition will be performed before the tagged elements;
- *Output interface:* is defined as the set of all its flow objects tagged with an output composition tag. Implicitly, it defines all the places of a business process

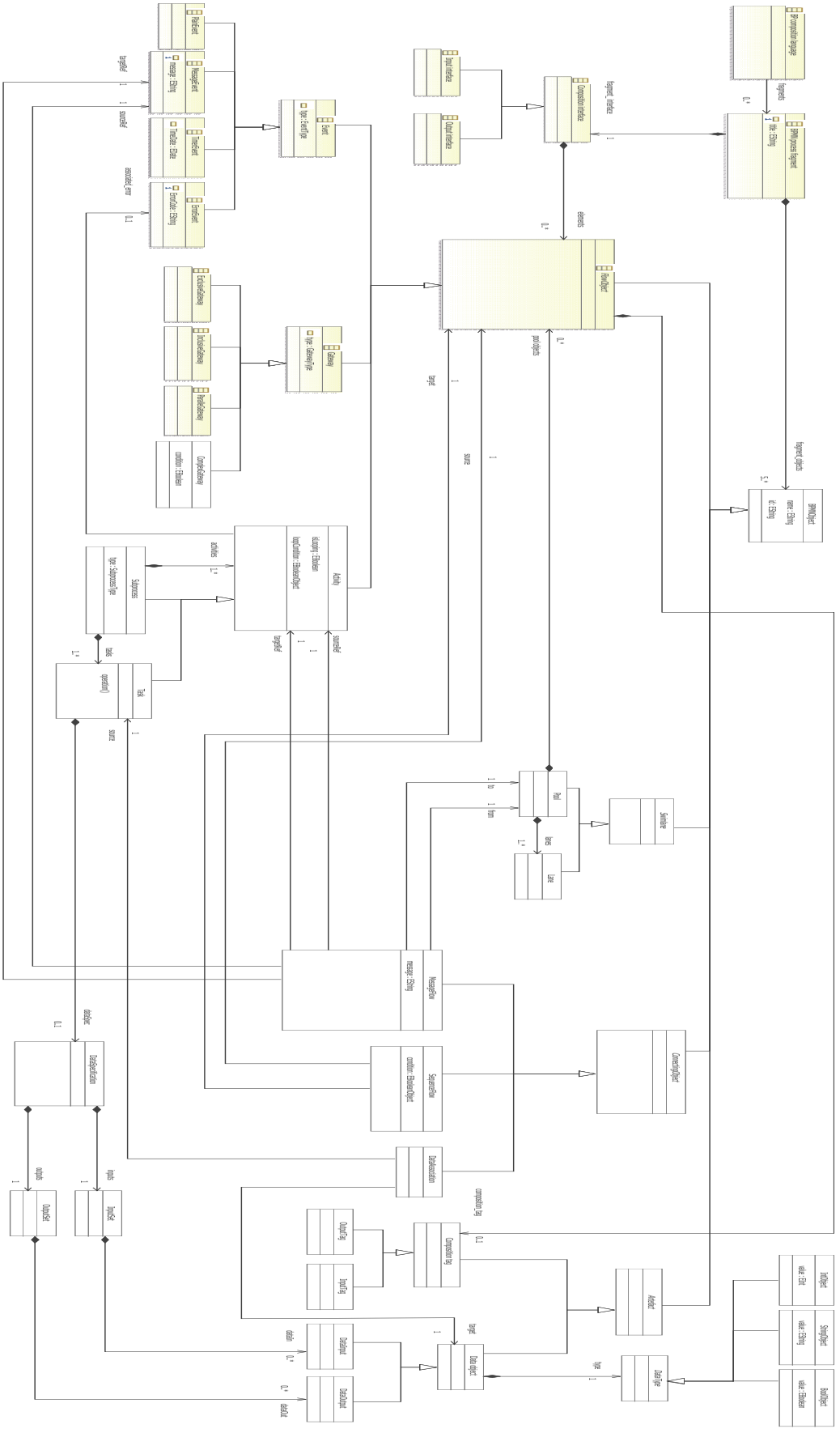


Fig. 4.7: Meta-model of composable business process fragment modelling language

fragment where, during the composition process, the actual composition will be performed after the tagged elements.

In order to have a complete and global view of the abstract syntax of the language and to understand all the relations and dependencies between the elements, the entire language meta-model is graphically depicted in Figure 4.7.

We want to establish and ensure that no ill-formed business process fragment models can be produced given the language meta-model. In other words, we want to be sure of the well-formedness of all the model instances that can be created with the proposed language. It is imperative to check the correspondence between the models and the language in which the models are written. To be sure that the business process fragments that can be created with the proposed language, we will check the alignment between the created models and a set of structural properties that any model of the same type must respect.

In model driven engineering, a meta-model is typically not refined enough to provide all the relevant aspects of a specification. There is a need to describe additional constraints about the objects in the model. The approach we follow is to express a set of desired well-formedness constraints in the Object-Constraint Language with respect to the meta-model of the business process fragment modelling language. The Object Constraint Language (OCL) [Gro06b] is a formal language that remains easy to read and write. It provides a formal language for specifying constraints which can supplement the models and meta-models created using MDE principles.

These consistency rules serve also for verifying the structural correctness of business process fragments. Therefore, they are presented in detail and discussed in Chapter 5.

4.2.3 Language support for composing business process fragments

As defined until now, our language allows the modelling of business process fragments and adds the notion of "composability" through the introduction of composition interfaces. The main goal of business process fragments is to be composed together for creating complete and more complex business processes. Business process composition is regarded as a flexible mechanism capable to cope with the increasing complexity of business processes. Similar to component-based software development, the core idea is to create a complex business process by assembling simpler ones - in our case, business process fragments. The complexity of building a business process is taken away from the business analyst and delegated to the actual composition. Another strong argument motivating the use of process composition is process reuse.

The general approach when applying model composition is to provide *composition operators*. They are mechanisms that take two (or more) models as input and generate an output that is their composition. Most languages provide a fixed set of composition operators, with explicit notations, specific behaviour and defined semantics. In case a language does not provide a composition operator with the desired behaviour, different workarounds need to be used. Therefore, we consider imperative to enrich our language with a set of well-defined composition operators, specifically defined for composing business process fragments.

All the composition operators we propose are *binary composition operators:* they take two business process fragments as input and produce a single process fragment as output of the composition. We propose the following composition operators:

- **Sequential composition operator:** is one of the most elementary composition operators we propose. This composition operator is used when there is a causality relation, either logical or functional, between the two business process fragments that are composed (one fragment cannot start until the other is over). As a basic condition, when applying this composition operator, the first fragment ($CBPF_1$) must be completed before the second ($CBPF_2$) can start. The result result of applying this operator is a business process fragment that performs (executes) the fragment ($CBPF_1$) first, followed by the fragment ($CBPF_2$), in sequence, one after the other.

  Requirements: to apply this operator, two conditions need to hold on the input business process fragments:

  - $CBPF_1$ has an *output composition interface* at one of its end events:
    $\exists\, e \in E_{e1}$ such that $e \in I_o$, where $E_{e1} = E_1 \cap \mathcal{E}_e$;
  - $CBPF_2$ has an *input composition interface* at its start event:
    $\exists\, e \in E_{s2}$ such that $e \in I_i$, where $E_{s2} = E_2 \cap \mathcal{E}_s$.

  Notation: $seq(CBPF_1, CBPF_2)$

  Composition interface of result: is the union of the interfaces of the input models, from which we need to remove $out(CBPF_1)$ and $in(CBPF_2)$:

  $I_{res} = I_{i-1} \setminus \{out(CBPF_1)\} \cup I_{i-2} \setminus \{in(CBPF_2)\}$

  For a better understanding, the semantics of this composition operator can be informally defined in terms of token passing, and is the following: a new token is generated at the start event of $CBPF_1$ and through the outgoing sequence flow arrives at the first flow element of $CBPF_1$, enabling it. Once the flow element has executed, the token is sent through to the next flow element. In the same manner, the token traverses in sequence all the flow elements of $CBPF_1$, then those of $CBPF_2$, until it reaches the end event of $CBPF_2$ where it is consumed. As this is the only token generated, the process is considered completed.

- **Parallel composition operator:** this operator represents the concurrent execution of two business process fragments. Two process fragments can be executed concurrently if they do not depend on each other, i.e., they are not causally linked. There is no communication between the two fragments that are composed. The result obtained when applying this composition operator performs the business process fragments ($CBPF_1$) and ($CBPF_2$) independently of each other (concurrently).

  Requirements: to apply this operator, two conditions need to hold on the input business process fragments:

  - Both $CBPF_1$ and $CBPF_2$ have an *output composition interface* at one of their end events:
    $\exists\, e_1 \in E_e1, e_2 \in E_e2$ such that $e_1 \in I_{o1}, e_2 \in I_{o2}$, where $E_e1 = E_1 \cap \mathcal{E}_e, E_e2 = E_2 \cap \mathcal{E}_e$;
  - Both $CBPF_1$ and $CBPF_2$ have an *input composition interface* at their start events:
    $\exists\, e_1 \in E_s1, e_2 \in E_s2$ such that $e_1 \in I_{i1}, e_2 \in I_{i2}$, where $E_s1 = E_1 \cap \mathcal{E}_s, E_s2 = E_2 \cap \mathcal{E}_s$.

  Notation: $par(CBPF_1, CBPF_2)$

Composition interface of result: contains the union of interfaces of the input models, from which we remove the start events and end events tagged with composition interfaces of $CBPF_1$ and $CBPF_2$, and add an output composition tag at the newly introduced end event and an input composition tag at the newly introduced start event:

$I_{res} = I_1 \cup I_1 \setminus \{in(CBPF_1), out(CBPF_1), in(CBPF_2), out(CBPF_2)\} \cup \{start_{new}, end_{new}\}$ where $start_{new} \in \mathcal{E}_s, end_{new} \in \mathcal{E}_e$

For a better understanding, the semantics of this composition operator can be informally defined in terms of token passing, and is the following: a token is generated by the start event and, through the outgoing sequence flow, reaches the parallel split gateway, that controls the diverging of the sequence flow. For this composition operator, two parallel flow are generated, one for each outgoing arc, and a token produced on each output flow of the gateway. The tokens traverse in parallel the two branches and are synchronized by the merging parallel gateway. After passing the merging gateway, the token is consumed by the end event.

- **Exclusive choice composition operator:** this operator is used to represent different possible paths of execution when the control flow is determined based on a specific condition or decision, or even non-deterministically. It also models non-determinism: the choice between the process fragments by default is made randomly. The result of applying the exclusive choice composition operator on two input fragments $CBPF_1$ and $CBPF_2$ is a business process fragment that can behave either like $CBPF_1$ or like $CBPF_2$. Once one the fragments ($CBPF_1$ or $CBPF_2$) executes its fist activity, the elements from the other fragments cannot be reached any more. This operator has basically two main use cases: presentation of alternate functionality, meaning that the main goal of the resulting fragment can be achieved in two (or more) distinct ways; another possibility refers to its use for representing fault tolerance.

  Requirements: to apply this operator, two conditions need to hold on the input business process fragments:

  - Both $CBPF_1$ and $CBPF_2$ have an *output composition interface* at one of their end events:
    $\exists\, e_1 \in E_e1, e_2 \in E_e2$ such that $e_1 \in I_{o1}, e_2 \in I_{o2}$, where $E_e1 = E_1 \cap \mathcal{E}_e, E_e2 = E_2 \cap \mathcal{E}_e$;
  - Both $CBPF_1$ and $CBPF_2$ have an *input composition interface* at their start events:
    $\exists\, e_1 \in E_s1, e_2 \in E_s2$ such that $e_1 \in I_{i1}, e_2 \in I_{i2}$, where $E_s1 = E_1 \cap \mathcal{E}_s, E_s2 = E_2 \cap \mathcal{E}_s$.

  Notation: $excl(CBPF_1, CBPF_2)$

  Composition interface of result: contains the union of interfaces of the input models, from which we remove the start events and end events tagged with composition interfaces of $CBPF_1$ and $CBPF_2$, and add an output composition tag at the newly introduced end event and an input composition tag at the newly introduced start event:

  $I_{res} = I_1 \cup I_1 \setminus \{in(CBPF_1), out(CBPF_1), in(CBPF_2), out(CBPF_2)\} \cup \{start_{new}, end_{new}\}$ where $start_{new} \in \mathcal{E}_s, end_{new} \in \mathcal{E}_e$

For a better understanding, the semantics of this composition operator can be informally defined in terms of token passing, and is the following: a token is generated by the start event and, through the outgoing sequence flow, reaches the exclusive split gateway, that controls the diverging of the sequence flow. A token is sent only on one of the output paths, based on the decision taken, activating just one of the two flows. The active path is then traversed by the token. The merge exclusive gateway must wait until the token from the active path arrives, and only then the sequence flow continues. After passing the merge gateways, the token is consumed by the end event.

- **Choice composition operator:** this operator is used to represent different possible paths of execution when the control flow is determined based on a specific condition or decision. It can be considered a special case of the exclusive choice composition operator. The result of applying the choice composition operator on two input fragments $CBPF_1$ and $CBPF_2$ is a business process fragment that can behave either like $CBPF_1$ or like $CBPF_2$ or like both of them. The particularity of this operator is that the business process fragments are executed alternatively, i.e., either one fragments is executed or the other, or both of them.

  Requirements: to apply this operator, two conditions need to hold on the input business process fragments:

  - Both $CBPF_1$ and $CBPF_2$ have an *output composition interface* at one of their end events:
    $\exists\, e_1 \in E_e1, e_2 \in E_e2$ such that $e_1 \in I_{o1}, e_2 \in I_{o2}$, where $E_e1 = E_1 \cap \mathcal{E}_e, E_e2 = E_2 \cap \mathcal{E}_e$;
  - Both $CBPF_1$ and $CBPF_2$ have an *input composition interface* at their start events:
    $\exists\, e_1 \in E_s1, e_2 \in E_s2$ such that $e_1 \in I_{i1}, e_2 \in I_{i2}$, where $E_s1 = E_1 \cap \mathcal{E}_s, E_s2 = E_2 \cap \mathcal{E}_s$.

  Notation: $cho(CBPF_1, CBPF_2)$

  Composition interface of result: contains the union of interfaces of the input models, from which we remove the start events and end events tagged with composition interfaces of $CBPF_1$ and $CBPF_2$, and add an output composition tag at the newly introduced end event and an input composition tag at the newly introduced start event:

  $I_{res} = I_1 \cup I_1 \setminus \{in(CBPF_1), out(CBPF_1), in(CBPF_2), out(CBPF_2)\} \cup \{start_{new}, end_{new}\}$
  where $start_{new} \in \mathcal{E}_s, end_{new} \in \mathcal{E}_e$

  For a better understanding, the semantics of this composition operator can be informally defined in terms of token passing, and is the following: a token is generated by the start event and, through the outgoing sequence flow, reaches the inclusive split gateway, that controls the diverging of the sequence flow. When the inclusive gateway is reached, for each outgoing sequence flow with a true condition, a token is generated and traverses that path. The merge inclusive gateway allows the process to continue only when tokens arrive from all incoming sequence flows where a token was generated before. After passing the merge gateway, the token is consumed by the end event.

- **Unordered (arbitrary) sequence composition operator:** two business process fragments can be either independent or logically correlated. This composition operator is usually applied in the case of total independence of the two fragments, when the unordered sequence operator becomes an alternative to the parallel composition operator. The operator can also be applied in the case when the process fragments are logically dependent, while being functionally independent, i.e., the two fragments complement each other. The result of applying the unordered sequence composition operator on two inputs business process fragments $CBPF_1$ and $CBPF_2$ is a fragment that performs either the behaviour specified by fragment $CBPF_1$ followed by fragment $CBPF_2$, or the behaviour of fragment $CBPF_2$ followed by $CBPF_1$ sequentially, but in no particular order.

  Requirements: to apply this operator, two conditions need to hold on the input business process fragments:

  - Both $CBPF_1$ and $CBPF_2$ have an *output composition interface* at one of their end events:
    $\exists\, e_1 \in E_e1, e_2 \in E_e2$ such that $e_1 \in I_{o1}, e_2 \in I_{o2}$, where $E_e1 = E_1 \cap \mathcal{E}_e, E_e2 = E_2 \cap \mathcal{E}_e$;
  - Both $CBPF_1$ and $CBPF_2$ have an *input composition interface* at their start events:
    $\exists\, e_1 \in E_s1, e_2 \in E_s2$ such that $e_1 \in I_{i1}, e_2 \in I_{i2}$, where $E_s1 = E_1 \cap \mathcal{E}_s, E_s2 = E_2 \cap \mathcal{E}_s$.

  Notation: $arb(CBPF_1, CBPF_2)$

  Composition interface of result: contains the union of interfaces of the input models, together with the previously created composition interface copies, from which we remove the start events and end events tagged with composition interfaces of $CBPF_1, CBPF_2, CBPF_1'$,
  $CBPF_2'$, and add an output composition tag at the newly introduced end event and an input composition tag at the newly introduced start event:

  $I_{res} = I_1 \cup I_2 \cup I_1' \cup I_2 \setminus \{in(CBPF_1), out(CBPF_1), in(CBPF_2), out(CBPF_2), in(CBPF_1'),$
  $out(CBPF_1'), in(CBPF_2'), out(CBPF_2')\} \cup \{start_{new}, end_{new}\}$ where $start_{new} \in \mathcal{E}_s, end_{new} \in \mathcal{E}_e$

  For a better understanding, the semantics of this composition operator can be informally defined in terms of token passing, and is the following: a token is generated by the start event and, through the outgoing sequence flow, reaches the exclusive split gateway, that controls the diverging of the sequence flow. When the exclusive gateway is reached, a single token is generated on one of the outgoing sequence flows, which activates one of the two paths which is then traversed. Thus, either all the flow objects of $CBPF_1$ be executed followed by those of $CBPF_2$, or they are executed in the order $CBPF_2$ then $CBPF_1$. The merge exclusive gateway allows the process to continue only when the tokens arrives from the active path. After passing the merge gateways, the token is consumed by the end event.

- **Parallel with communication composition operator:** this operator is absolutely necessary whenever two business process fragments are mutually dependent: during its operation, one fragment may require some data produced by the other and vice versa. The operator enhances the basic parallel composition one to operate

in the case where two concurrent fragments need to synchronize or exchange data during their execution. It models a functional dependence between the two process fragments that are composed, which prevents their sequential execution. The result of applying the parallel with communication composition operator on two input process fragments $CBPF_1$ and $CBPF_2$ is a business process fragment that performs the fragments $CBPF_1$ and $CBPF_2$ independently of each other (in parallel) - represents the concurrent execution of the fragments. Further more, the concurrent process fragments may synchronize and exchange information over a *set of communication elements* belonging to the two fragments.

Requirements: to apply this operator, some conditions need to hold on the input business process fragments:

- Both $CBPF_1$ and $CBPF_2$ have an *output composition interface* at one of their end events:
  $\exists\, e_1 \in E_e1, e_2 \in E_e2$ such that $e_1 \in I_{o\,1}, e_2 \in I_{o\,2}$, where $E_e1 = E_1 \cap \mathcal{E}_e, E_e2 = E_2 \cap \mathcal{E}_e$;
- Both $CBPF_1$ and $CBPF_2$ have an *input composition interface* at their start events:
  $\exists\, e_1 \in E_s1, e_2 \in E_s2$ such that $e_1 \in I_{i1}, e_2 \in I_{i2}$, where $E_s1 = E_1 \cap \mathcal{E}_s, E_s2 = E_2 \cap \mathcal{E}_s$;
- For each of the two fragments, there is at least one activity or message event inside the fragment tagged with an input or output composition tag. These tagged elements make up the *set of communication elements* (SCE) for the two fragments. The SCE is a set of pairs of the form (x,y), where: x denotes an activity or message event belonging to $CBPF_1$ having a composition tag; y denotes an activity or message event belonging to $CBPF_2$ having a composition tag; the tags of x and of y must be different. The role of the composition tags on the elements from the SCE is to define the directionality of the message exchange. For example, for a pair (x,y) from the SCE, if x is tagged with an output composition interface and y with an input composition interface, then in the process fragment resulting from their composition, there will be a message exchange from x to y. Moreover, the number of elements (pairs) in SCE gives the number of message exchanges that will appear in the resulting fragment after the composition.

Notation: $parC(CBPF_1, CBPF_2)$

Composition interface of result: contains the union of interfaces of the input models, from which we remove the start events and end events tagged with composition interfaces of $CBPF_1$ and $CBPF_2$, and add an output composition tag at the newly introduced end event and an input composition tag at the newly introduced start event:

$I_{res} = I_1 \cup I_1 \setminus \{in(CBPF_1), out(CBPF_1), in(CBPF_2), out(CBPF_2)\} \cup \{start_{new}, end_{new}\}$
where $start_{new} \in \mathcal{E}_s, end_{new} \in \mathcal{E}_e$

- **Refinement composition operator:** *refinement* is the transformation of a design from a high level abstract form to a lower level more concrete form, hence allowing hierarchical modelling. In our case, the refinement operation consists in replacing an activity from a business process fragments by a more refined construct (another

process fragment) in order to introduce a higher level of detail in the initial business process fragment. The result of applying the the refinement composition operation on two input business process fragments $CBPF_1$ and $CBPF_2$, at the specific activity a, behaves as fragment $CBPF_1$, except for the activity a, which is replaced by the process fragment $CBPF_2$.

Requirements: to apply this operator, some conditions need to hold on the input business process fragments:

- Fragment $CBPF_1$ must have an an *input* or *output* composition interface at one of its activities:
  $\exists\, a \in A_1$ such that $a \in I_{i1} \vee a \in I_{o1}$;
- Fragment $CBPF_2$ must have an *output composition interface* at its start event and an *input composition interface* at one of its end events:
  $\exists\, e_1 \in E_s2, e_2 \in E_e2$ such that $e_1 \in I_{o2}, e_2 \in I_{i2}$, where $E_s2 = E_2 \cap \mathcal{E}_s, E_e2 = E_2 \cap \mathcal{E}_e$.

Notation: $ref(CBPF_1, CBPF_2)$

Composition interface of result: contains the union of interfaces of the input models, from which we remove the tagged activity of $CBPF_1$ and the start and tagged end event of $CBPF_2$:

$I_{res} = I_1 \cup I_2 \setminus \{comp1, in(CBPF_2), out(CBPF_2)\}$

For a better understanding, the semantics of this composition operator can be informally defined in terms of token passing, and is the following: a new token is generated at the start event of $CBPF_1$ and, through the outgoing sequence flow, arrives at the last flow element of $CBPF_1$ before activity *comp1*. Once that flow element has executed, the token is sent through to the first flow element of $CBPF_2$. It then traverses all the flow objects of $CBPF_2$, until it reaches the end event of $CBPF_2$ where it is consumed.

- **Synchronization composition operator:** specifies a situation in which two business process fragments synchronize their execution because they have specific similarities between one or more of their flow objects. Synchronization can only be done at the level of activities of a process fragment. Therefore, for performing the actual synchronization operation, a matching process should be performed first that determines the set of activities that match between the two process fragments. This set is called the *synchronization set Sync*, where the actual synchronization operation will be performed. The synchronization set contains activities from the two process fragments that match and where the actual synchronization will be performed. The following restriction applies: it is only possible to synchronize activities from one fragment with activities in the other fragment. The result of applying the synchronization composition operator on two input business process fragments $CBPF_1$ and $CBPF_2$, at the specific locations specified by the synchronization set Sync, is a business process fragment that performs in parallel (concurrently) the parts of the two process fragments which do not belong to the synchronization set Sync, and merges (synchronizes) and performs only once the elements from the synchronization set.

  Requirements: to apply this operator, some conditions need to hold on the input business process fragments:

  - There must exist a synchronization set: $\exists\, Sync = \{(x, y) | x \in F_1, y \in F_2\}$;

- Fragments $CBPF_1$ and $CBPF_2$ have flow objects (activities) belonging to the synchronization set Sync: $\exists\, x \in F_1, y \in F_2$ such that $(x, y) \in Sync$;
- The flow object of fragment $CBPF_1$ that belong to the synchronization set Sync must have an *input composition tag*: $(x, y) \in Sync, x \in F_1 \Longrightarrow x \in I_{i1}$;
- The flow object of fragment $CBPF_2$ that belong to the synchronization set Sync must have an *output composition tag* $(x, y) \in Sync, y \in F_2 \Longrightarrow y \in I_{o2}$;

Notation: $sync(CBPF_1, CBPF_2)$

Composition interface of result: is the union of the ones of the input process fragments, from which we remove the elements belonging to the synchronization set Sync, and add an input composition tag on the new start event and an output composition tag on the new end event:

$I_{res} = I_1 \cup I_2 \setminus \{x, y | (x, y) \in Sync, x \in A_1, y \in A_2\} \cup \{start_{new}, end_{new}\}$, where $Tag(start_{new}) \in CT_i$, $Tag(end_{new}) \in CT_o$

For a better understanding, the semantics of this composition operator can be informally defined in terms of token passing, and is the following: a new token is generated at the start event and through the outgoing sequence flow reaches the first split parallel gateway. Here, two tokens are generated for each outgoing sequence flow, which are executed in parallel. The flows synchronize at the merge parallel gateway. From its output sequence flow, the token is passed to the first merged synchronization element. Further on, the same idea is applied for the process areas situated between and bellow synchronization elements, as they are put in parallel and tokens traverse them. After the last merge parallel gateway, the token is consumed by the end event.

- **Insertion composition operator:** this composition operator is inspired from the "insert process fragment" pattern, belonging to the workflow design patterns [vdAtH99]. The application of the insertion composition operator on two input business process fragments $CBPF_1$ and $CBPF_2$ consists in inserting the business process fragment $CBPF_2$ *before* or *after* a certain activity of process fragment $CBPF_1$. The application of this composition operator requires the explicit marking of the activity where the insertion is performed. This is done with the help of the composition tags: the activity where the insertion will be performed is explicitly marked with either an input or an output composition tag.

  Two separate cases of insertion are possible, depending on the type of the composition tag:

  - an input composition tag implies that the insertion will be performed before/above the tagged activity;
  - an output composition tag implies the insertion will be performed after/below the tagged activity.

  Requirements: to apply this operator, some conditions need to hold on the input business process fragments:

  - Fragment $CBPF_1$ must have either an input or an output composition tag at one of the activities within the fragment:
    $\exists\, a \in A_1$ such that $a \in I_{i1} \vee a \in I_{o1}$; ;

- Fragment $CBPF_2$ must have an input composition tag at the start event and an output composition tag at one of the end events:
  $\exists\, e_1 \in E_s2, e_2 \in E_e2$ such that $e_1 \in I_{o2}, e_2 \in I_{i2}$, where $E_s2 = E_2 \cap \mathcal{E}_s, E_e2 = E_2 \cap \mathcal{E}_e$;
- For ease of use, we denote by *act1* the activity from fragment $CBPF_1$ tagged with a composition interface.

Notation: $ins(CBPF_1, CBPF_2)$

Composition interface of result: is the union of those of the input process fragments, from which we remove the tagged activity of fragment $CBPF_1$ and the start and tagged end event of fragment $CBPF_2$. We then add an input composition tag on the start event of the result and an output composition tag on the end event:

$I_{res} = I_1 \cup I_2 \backslash \{act1, in(CBPF_2), out(CBPF_2)\} \cup \{Es_{res}, Ee_{res}\}$, where $Es_{res} = Es_1, Ee_{res} \in Ee_1$ and $Tag(Es_{res}) \in CT_{in}$, $Tag(Ee_{res}) \in CT_{out}$

For a better understanding, the semantics of this composition operator can be informally defined in terms of token passing, and is the following: a new token is generated at the start event of fragment $CBPF_1$ and through the outgoing sequence flow reaches the last flow object before the activity tagged with a composition interface. In case of an insert before composition, the token then passes to the first flow object from $CBPF_2$ and afterwards through all the flow objects of $CBPF_2$.It then arrives at the tagged activity of $CBPF_1$ and passe the rest of the flow object of $CBPF_1$, before reaching the end event where the token is consumed. In case of an insert after composition, the token goes to the tagged activity of $CBPF_1$, followed by all the flow objects of $CBPF_2$. It then continues with the successor of the tagged activity of $CBPF_1$ and follows with the rest of the flow objects of $CBPF_1$, before reaching the end event where it is consumed.

The set of composition operators designed specifically for the composition of business process fragments are discussed in detail in Chapter 7. In the same chapter they are also defined in a formal manner using a set-based mathematical specification.

However, these operators are not yet part of the CBPF domain specific language. Therefore, in the following, we enrich our language with a set of well-defined composition operators. They are added to the abstract syntax of the language. We do this by extending the language's meta-model with the appropriate support for the composition operators.

In Figure 4.8 we present an excerpt of the business process fragment modelling and composition language meta-model, which presents the newly introduced composition operators. The central meta-class of this meta-model is *BP composition language*. It denotes the business process fragment modelling and composition language that we propose throughout this chapter. As it can be seen from the diagram, the language allows to create several *Composable business process fragments*. They have been discussed in detail in the previous sections. Therefore, the meta-model in Figure 4.8 shows only some high level details regarding the modelling of business process fragments. A composable business process fragment contains several fragment objects (*CBPF object*). As we saw previously, there are four main classes of elements that can appear in a business process fragment: *flow objects*, *swimlanes*, *connecting objects* and *artifacts*. Moreover, a business process fragment also has a *composition interface*.

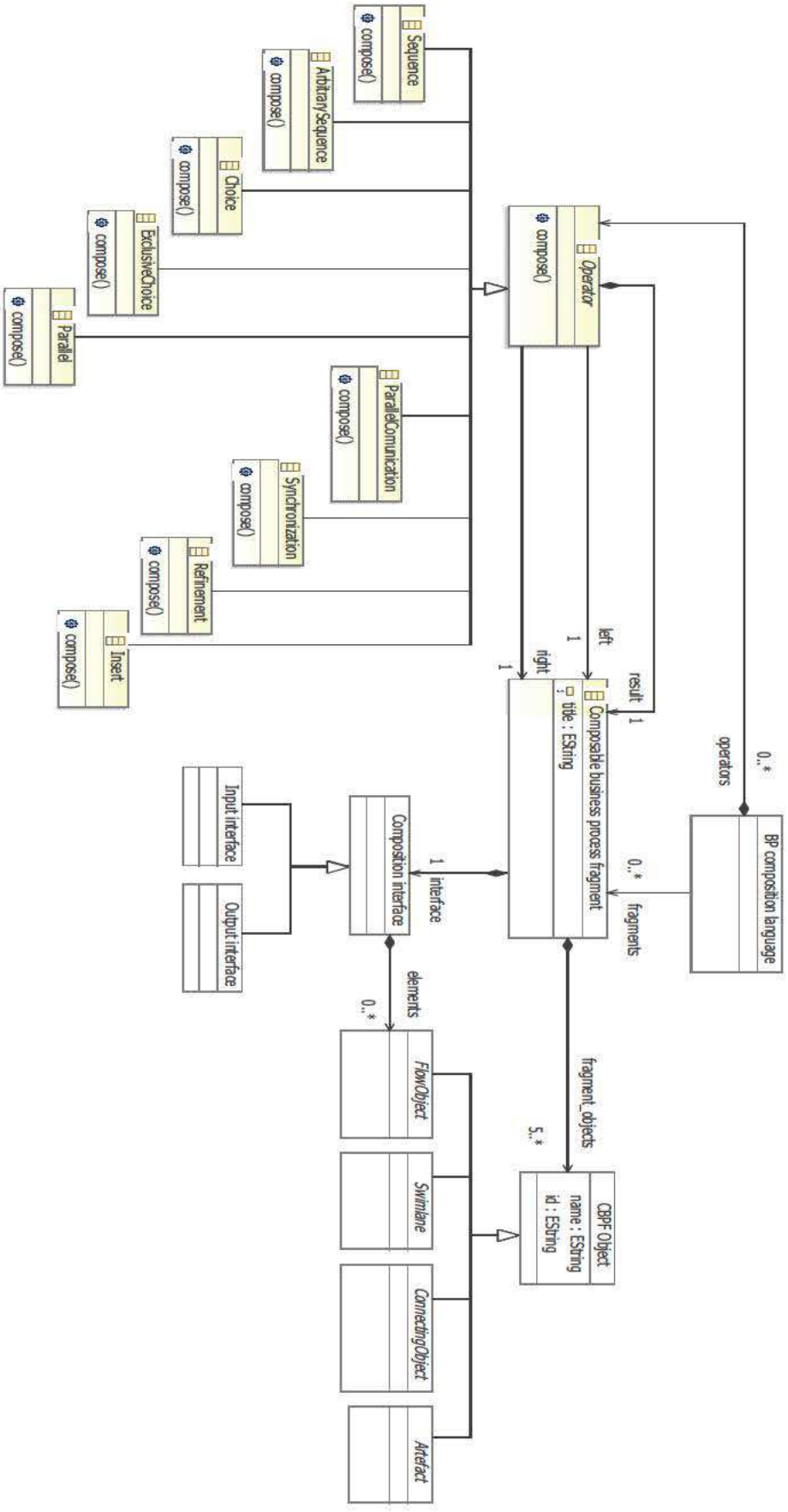


Fig. 4.8: Excerpt of language meta-model: composition operators

In the context of business process composition and composition operators, it is the left part of Figure 4.8 that is more relevant. The language contains a set of *composition operators*. This is described in the meta-model by the containment relation between the *BP composition language* and the *Operator* meta-classes. The *Operator* meta-class is an *abstract* one. Its purpose is to subsume several possible concrete meta-classes, which define specific composition operators and which will override the abstract meta-class. The *Operator* meta-class contains an abstract operation called *compose()*, which will implement the actual composition. As presented earlier, all the operators that we propose are *binary composition operators*. This is also represented on the meta-model. The *Operator* meta-class is connected through two reference relations to the *Composable business process fragment* meta-class. This denotes the fact that every composition operator has two *operands*: a *left* one and a *right* one. There is a third relation between the two meta-classes, denoting the fact that the *result* of a composition is also a business process fragment. However, the type of relation is this time different: *containment relation*. We use it in order to specify that a new business process fragment is created as a result of a composition operation. Moreover, the result of a composition can further be used in other composition as an operand.

The *Operator* meta-class is an abstract one and only defined at a high level the composition operators. Using the *inheritance relation*, we add nine new meta-classes which denote concrete composition operators:

- *Sequence:* defines the sequential composition operator;
- *ArbitrarySequence:* represents the unordered (arbitrary) sequence composition operator;
- *Choice:* denotes the choice composition operator;
- *ExclusiveChoice:* denotes the exclusive choice composition operator;
- *Parallel:* defines the parallel composition operator;
- *ParallelComunication:* defines the parallel with communication composition operator;
- *Synchronization:* denotes the synchronization composition operator;
- *Refinement:* denotes the refinement composition operator;
- *Insertion:* denotes the fragment insertion composition operator.

Each of these meta-classes contains an operation called *compose()*. This operation overrides the one with the same name defined in the *Operator* meta-class. Each individual operation thus implements the specific type of composition defined the composition operator in cause.

4.2.4 Language support for product derivation specification

The last step of the SPL methodology we proposed in Chapter 3, called *product derivation specification*, takes as input a set of business process fragments and transforms them, using a compositional approach, into a proper business process that models the behaviour of the SPL product being derived. The steps involved in the product derivation specification process are the following:

- Annotation of business process fragments with composition interfaces;
- Creation of the composition workflow;
- Selection of applied composition operators;
- Iterative application of composition operators.

We consider that the CBPF language proposed throughout this chapter should also offer the necessary support for the creation of such product derivation specifications. For the first step of the process, the annotation of business process fragments with composition interfaces, the necessary support is already provided: we can create new business process fragments and define their composition interfaces. For the actual composition, we can use the set of composition operators that are already available in the language. However, the creation of the composition workflow is not supported with the current version of our language. The role of the composition workflow is to specify the exact order in which the business process fragments are composed. It also specifies the exact composition operators that will be applied.

As defined in Chapter 3, a composition workflow has the following elements:

- Fragment place-holders: for the composition workflow, business process fragments are seen as black boxes, we are not interested in their internal representations;
- Operators: the goal of the composition workflow is to specify the exact order in which process fragments are composed. It is essential to to be able to represent the different types of business process composition operators that can be applied;
- Connectors: we need to be able to represent the sequencing/flow of elements in the composition workflow.

Therefore, we need to extend the CBPF language with the necessary support for creating such composition workflows. In Figure 4.9 we present a part of the language meta-model that defines this language extension. The main class of the meta-model is *BP composition language*, which denotes the language we are proposing and defining throughout this chapter. The language allows the creation of several *composable business process fragments*. This part is not detailed, as it was introduced previously at the beginning of the chapter. The language also allows to specify different derivation workflows (meta-class *ProductDerivationSpecification*). Such a product derivation specification is characterized by a unique name. Each product derivation specification contains one or more elements (meta-class *PDSObject*). This is modelled using a containment relation between the two meta-classes. Each such object is characterized by a unique *id*. We define three types of product derivation specification objects:

- *Operators:* this meta-class defines the composition operators previously introduced in the language. It has been discussed in detail in the previous subsection. All the operators are binary ones and produce as a result a new composable business process fragment.

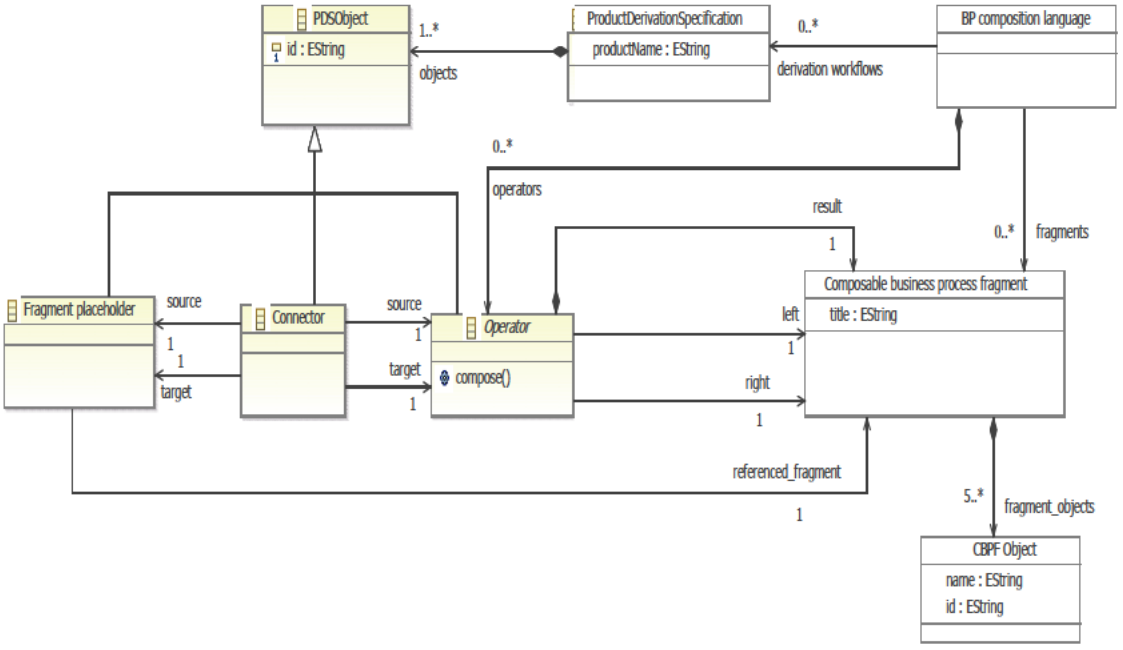


Fig. 4.9: Excerpt of language meta-model: support for product derivation specification

- *Fragment place-holders:* in order to maintain simplicity and ease of use, in the composition workflow, business process fragments are seen as black boxes, we are not interested in their internal representations. Therefore, in order to reduce complexity, a composition workflow contains fragment place-holders instead of the actual business process fragments that are composed. Each fragments place-holder has a string attribute *fragName* which corresponds to the name of the actual fragment which they substitute. A fragment place-holder references an actual business process fragment for further use (the actual composition). This is specified through a reference relation between the *Fragment placeholder* and *Composable business process fragment* meta-class.

- *Connectors:* for a complete specification, we need to be able to represent the sequencing/ordering of elements in the composition workflow. That is why we use simple directed connectors. A connector has a single *source* and a single *target* which can be fragment place-holders or operators. It will thus connect a fragment place-holder to an operator or vice-versa.

It should be noticed that, although defined in the same meta-model and therefore part of the same language, *composable business process fragments* and *composition workflows* represent *two different types of diagrams*, situated at two different levels of abstraction. While a composable business process fragment describes the "insides" and is interested in the concrete implementation and functioning of a business process fragment, a composition workflow looks at business process fragments as black-boxes, and is thus not interested in their exact functioning and sees them from a higher level of abstraction. Therefore, concepts belonging to these two types of models (diagrams) that can be created with the CBPF language should not be mixed together.

This concludes the presentation of the abstract syntax of the CBPF domain specific language. In the next section, we propose a concrete syntax for our language, that will enable language users to graphically represent composable business process fragments that conform to the CBPF meta-model.

4.3 Concrete graphical syntax

The *concrete syntax* defines the actual representation of the CBPF models. A concrete syntax acts as an interface between the instances of the concepts, and the human being supposed to produce or read them. It defines the physical appearance of our domain specific language. For a graphical language like CBPF, this means that it defines the graphical appearance of the language concepts and how they may be combined into a model.

Almost each of today's modelling languages comes with a graphical representation in order to improve readability and usability. Thus, the concrete syntax of modelling languages should be defined in terms of a visual language. It describes a set of visual sentences which in turn are given by a set of visual elements. A visual element can be seen as an object characterized by values of some attributes. We therefore propose a graphical concrete syntax for our CBPF language.

4.3.1 Direct definition of graphical concrete syntax

Most solutions to graphical concrete syntax definition on top of a meta-model are based on ad-hoc symbol editors. Early domain-specific modelling tools such as Meta-Case's MetaEdit+ [Poh03] or GME [Dav03] derive the structure of the graphical representation from the abstract syntax, as notation definitions are assigned directly for each abstract syntax model element. For each representable element of the meta-model, one defines an icon and indicates properties to be displayed. We make use of this approach for defining an initial form of the concrete graphical syntax of our language.

In the following we detail the graphical representation of the concepts of the language. For this purpose, we take all the elements defined in the abstract syntax and for each one indicate the corresponding graphical representation. As was discussed during the definition of the abstract syntax, the language we propose contains a subset of the most relevant elements found in the BPMN standard. As BPMN is a standard for modelling business processes and well known by the industrial and research communities, users are familiar with its notation. Therefore, we use a similar graphical notation as the one proposed by BPMN for the elements that are common to both languages.

All **events** share the same shape footprint, which is a *small empty circle*. Different line styles distinguish between the three types of events. In order to further distinguish between the different triggers that an event might have, specific representative icons can be included within the shape.

- *Start:* is graphically represented by a circle drawn with a *single thin line*;
- *End:* is graphically represented by a circle drawn with a *single thick line*;
- *Intermediate:* is graphically represented by a circle drawn with a *double thin black line*;

- *Plain:* the modeller does not graphically display the type of event. Therefore, all plain events (start, intermediate or end) are graphically displayed as regular start, intermediate respectively end events;
- *Message:* the trigger is displayed with a specific *graphical marker*, in this case an *envelope*, set inside the empty circle;
- *Timer:* the trigger is displayed with a specific *graphical marker*, in this case a *clock*, set inside the empty circle;
- *Error:* the trigger is displayed with a specific *graphical marker*, in this case a *a lightning bolt*, set inside the empty circle.

Activities are graphically represented by an *empty rectangle with rounded corners.* There are several possible types of activities:

- *Task:* is graphically represented by a *rectangle that has rounded corners* which must be drawn with a single thin black line;
- *Collapsed sub-process:* is graphically represented by a r*ounded corner rectangle* that must be drawn with a single thin black line. In order to differentiate between collapsed and expanded sub-processes, the collapsed sub-process contains a specific *marker.* The marker is be a *small square with a plus sign inside.* The square is positioned at the bottom center of the shape;
- *Expanded sub-process:* is graphically represented by a r*ounded corner rectangle* that must be drawn with a single thin black line. It does not contain any specific marker.

Gateways are graphically represented with the diamond symbol, drawn with a single thin black line. The symbol is used as it has been used in many flow chart notations for exclusive branching and is familiar to most modellers. To differentiate between the four different types of gateways, we use markers which are placed inside the diamond symbol.

- *Parallel:* uses a marker that is in the shape of a plus sign and is placed within the gateway diamond symbol to distinguish it from other gateways;
- *Exclusive:* no specific marker is required for this type of gateway;
- *Inclusive:* uses a marker that is in the shape of a circle or an "O", placed within the gateway diamond shape to distinguish it from other Gateways;
- *Complex:* uses a marker that is in the shape of an asterisk and is placed within the gateway diamond shape to distinguish it from other gateways.

The graphical representations for all the flow objects (events, activities and gateways) are displayed in Figure 4.10.

Pools: from a graphical point of view, a pool is a container for partitioning a process fragment. It is graphically represented by a square-cornered rectangle that must be drawn with a solid single black line. A pool will extend the entire length of the diagram, either horizontally or vertically. However, there is no specific restriction to the size and/or positioning of a pool.

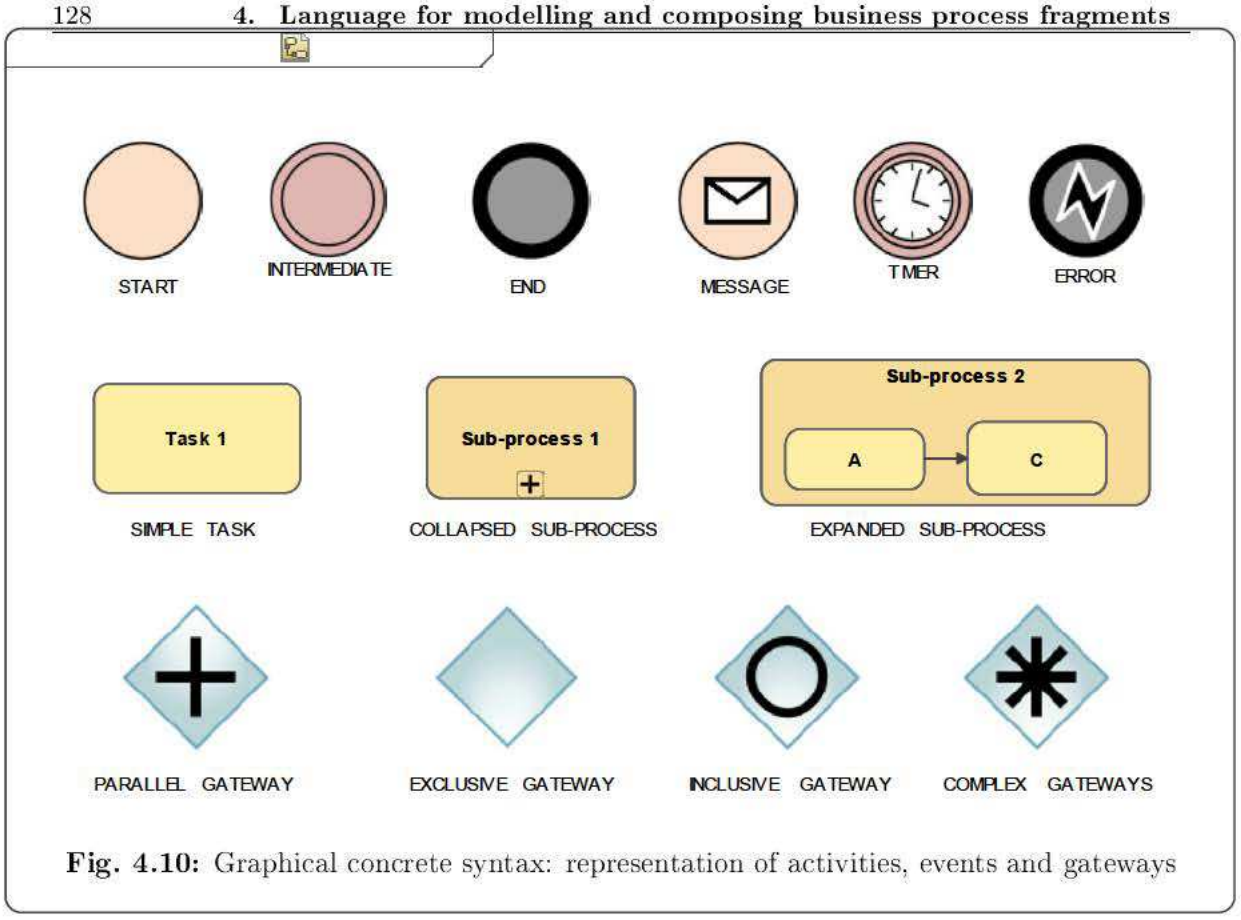


Fig. 4.10: Graphical concrete syntax: representation of activities, events and gateways

Lanes: are sub-partitions of a pool and will extend the entire length of the pool, either vertically or horizontally. Graphically, lanes are represented in the same manner as pools: a square-cornered rectangle that must be drawn with a solid single black line. However, lanes can only appear inside a pool, which contains them.

Sequence flow: the graphical representation of a sequence flow is a line with a solid arrowhead that must be drawn with a solid single line. The directionality of the sequence flow is from the source flow object towards the target flow object. A sequence flow can have a conditional expression attached to it, expressed as a boolean attribute. In this case, the condition is graphically represented as a text label attached to the arrow representing the sequence flow.

Message flow: graphically, a message flow is a line with an open arrowhead that MUST be drawn with a *dashed single black line.*

Data association: graphically, a data association is a line that must be drawn with a dotted single black line. It also has a single simple arrowhead at one of its ends. The directionality of the data association is given by the presence of the arrowhead at one of the ends of the dashed line. When the data associations connects an input data object with a task, the arrowhead points towards the task. In case an output data object is connected with a task, the arrowhead points towards the data object.

Composition tags: identify an exact place in a business process fragment where the fragment can be composed with other fragments. A composition tag is simply a *text annotation in form of a stereotype* that can appear on elements of a business process fragment. Composition tags are added on flow objects of a business process fragment. From a graphical point of view, a composition tag is a textual stereotype that is attached to a flow object belonging to the business process fragment. Depending on the type of composition tag (input

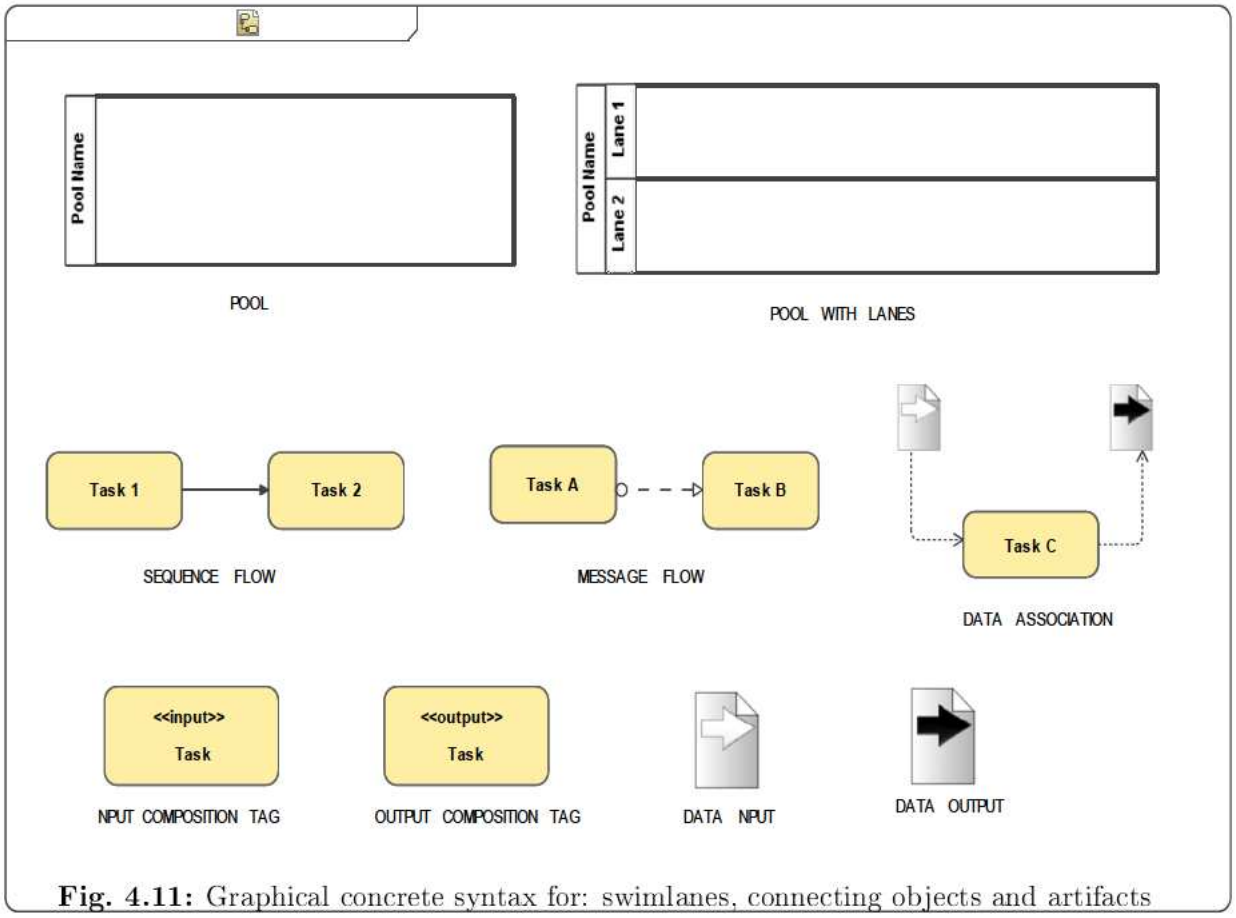


Fig. 4.11: Graphical concrete syntax for: swimlanes, connecting objects and artifacts

or output), we have two different stereotypes that can be applied: $\langle\langle input \rangle\rangle$ or $\langle\langle output \rangle\rangle$.

Data objects: graphically, a data object is depicted as a portrait-oriented rectangle that has its upper-right corner folded over, and must be drawn with a solid single black line. However, we must be able to graphically differentiate between the two existing types of data objects: input and output. This distinction is made by the addition of an empty or filled arrow at the top of the rectangle representing the data object. The *empty arrow* thus denotes an input data object, while the presence of the *filled arrow* denotes an output data object.

The graphical representations for swimlanes (pools and lanes), connecting objects (sequence flow, message flow, data association) and artifacts (composition tags and data objects) are displayed in Figure 4.11.

The graphical concrete syntax presented until now corresponds to the part of the meta-model (abstract syntax) that allows the modelling of composable business process fragments. However, we also need to add a graphical syntax for the part of the meta-model that defines the product derivation specification and the business process fragment composition operators.

Composition operators: are using for composing business process fragments. In one of the previous sections, we proposed a set of 9 such composition operators. From a graphical point of view, a composition operator is depicted as an *equilateral triangle* that has one of its tips pointing to the left, and must be drawn with a solid single black line. However, as there are 9 different types of composition operators, we need to be able to graphically differentiate between them. Therefore, we propose to add a simple text label inside the

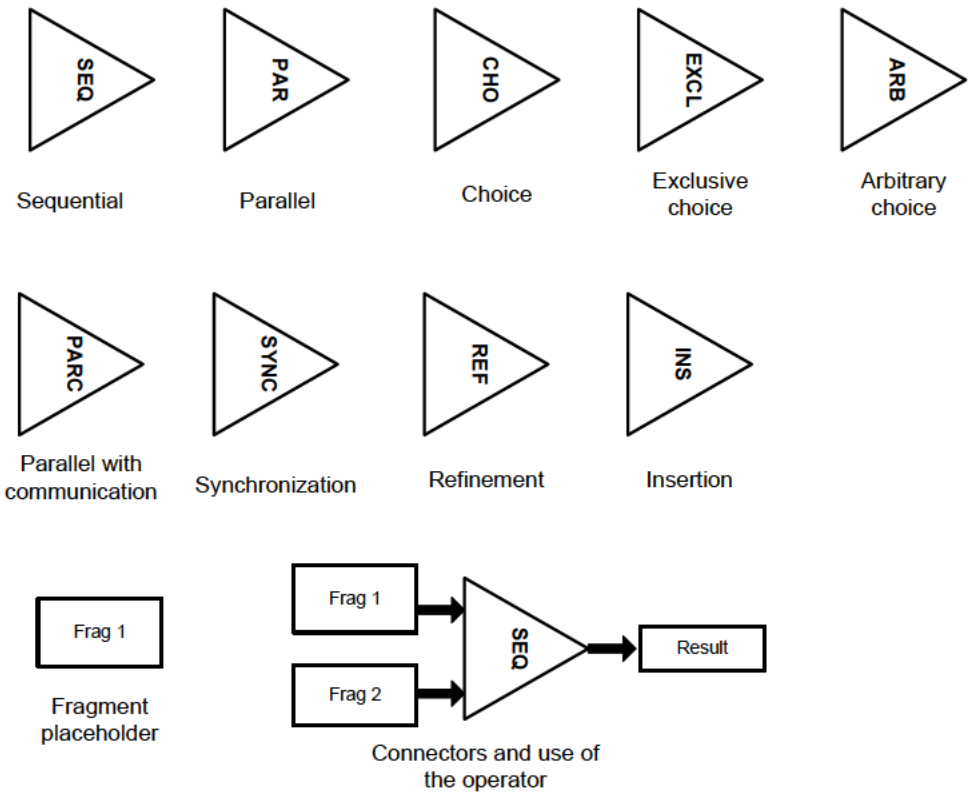


Fig. 4.12: Graphical concrete syntax for product derivation specification elements

triangle shape that will characterize each individual composition operator. The text labels used are the following:

- *Seq* for the sequential operator;
- *Par* for the parallel operator;
- *Cho* for the choice composition operator;
- *Excl* for the exclusive choice composition operator;
- *Arb* for the arbitrary (unordered) sequence operator;
- *ParC* for the parallel with communication operator;
- *Sync* for the synchronization operator;
- *Ref* for the refinement operator;
- *Ins* for the insertion composition operator.

Fragment place-holders: are black-box views of business process fragments and are used for simplicity reasons in the composition workflows of the product derivation specification as place-holders or substitutes for the actual business process fragments. Graphically, they are represented as a simple *rectangle with sharp edges* drawn with a continuous black line.

Connectors: are used for connecting together the different elements of a product derivation specification. A connector has a single source and target. They are used for creating the sequencing order for the composition operations. We use the same graphical representation for connectors as the one introduced for the *sequence flow*: a continuous black line with a filled black arrow at the target end.

The graphical representations for the different product derivation specification elements: composition operators, fragment place-holders and connectors are displayed in Figure 4.12.

4.3.2 Meta-model based graphical concrete syntax

A straightforward strategy to balance abstraction with expressive power is to separate abstract and concrete syntax representations. Essentially, this approach treats the visual notation as a separate language with its own element types, attributes and relations, on an additional modelling layer. For two dimensional graph-like languages, this visualisation grammar is derived from a core diagram meta-model, which contains attributed nodes and edges. By refining these concepts to specific model elements, the structure of the concrete syntax may be elaborated.

In [CSW08] Clark et al. propose a meta-model based way of describing the concrete syntax of a language and how to connect it with the abstract syntax. In order to describe how to interpret a diagram, they first define what it means to be a diagram at some level of abstraction. This is achieved by creating a model of diagrams in XMF, presented in detail in [CSW08]. This model is very similar with OMG's diagram interchange model. This enables it to capture the concrete syntax concepts, and the relationship between them, for a broad spectrum of diagram types, ranging from sequence diagrams to state machines to class diagrams.

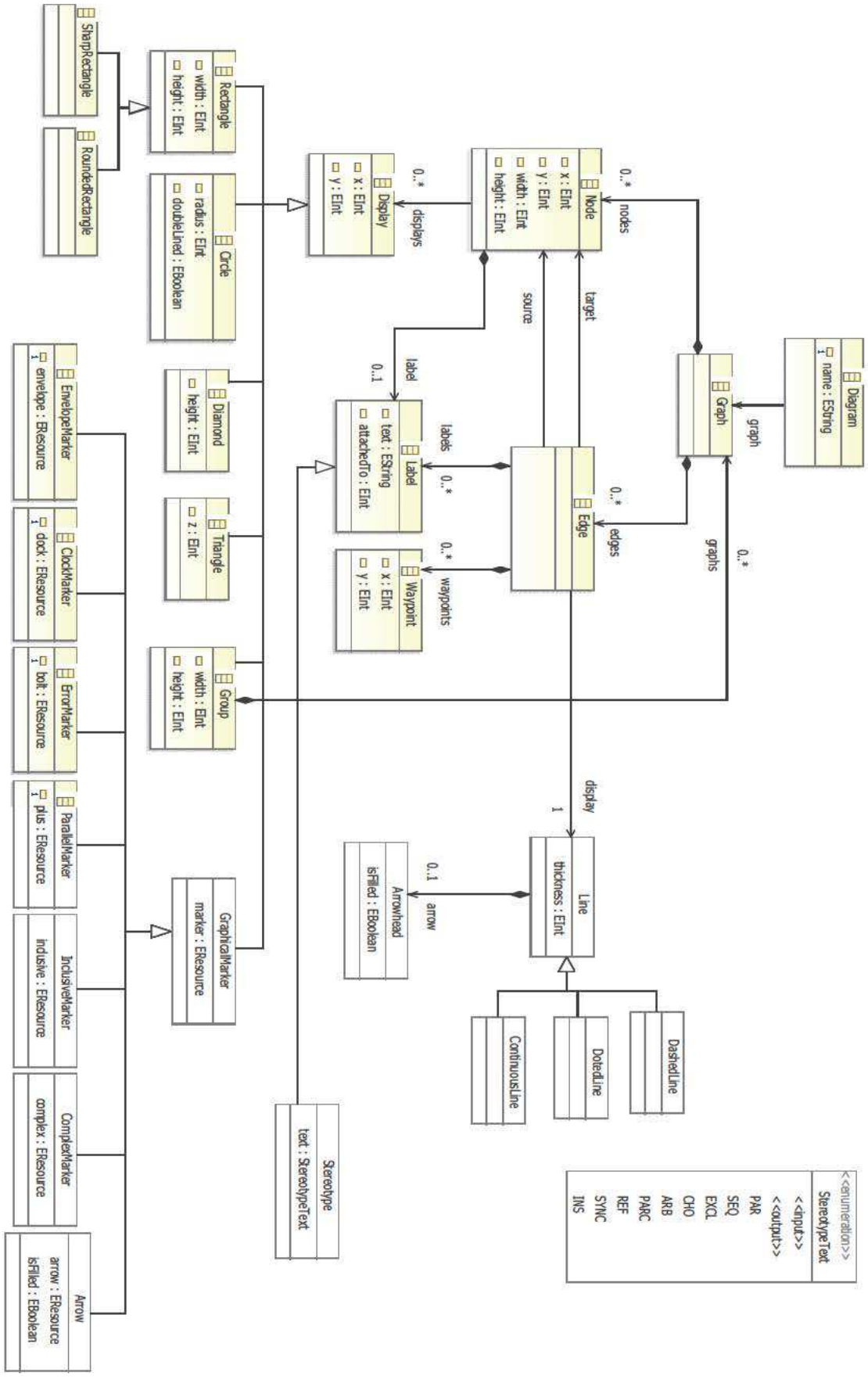


Fig. 4.13: A general model of diagrams for describing the concrete syntax of the language

We adapt this model for the specific needs of our concrete syntax. The resulting model is presented in Figure 4.13. The main meta-class of the meta-model is *Diagram* and defines in general all the diagrams that can be represented with the language. All diagrams are represented as *graph-like structures*. *Graphs* contain two main types of elements: *nodes* and *edges*. *Nodes* are used for representing the main graphical elements in a diagram, while the role of *edges* is to connect different nodes.

A *node* is characterized by four integer type attributes: the x and y coordinates give the position of the node on the screen; the *width* and *height* attributes characterize the different graphical shapes that a node may be represented by. A *node* is however an abstract concept, which requires different graphical shapes that will represent it in the diagram. Therefore, in the meta-model, a node has several *displays*. These displays are the concrete graphical shapes that are used for representing a node in the diagram. They are adapted to the particular needs of the graphical concrete syntax we are creating. We have seven main types of displays possible, represented in the meta-model as sub-classes of the *display* meta-class:

- *Rectangle:* this basic shape describes a simple rectangle. It is characterized by two attributes: the *width* and the*height*, which have both integer values. Due to the specific needs of our graphical syntax, we distinguish between rectangles that have *sharp edges* and those that have *rounded edges*.
- *Circle:* this basic shape describes a simple circle. It inherits the x and y attributes from the *Display* super-class which are used to define the position of the center of the circle on the screen. Moreover, there is another integer attribute called *radius* that defines the exact radius of the circle, so it can be drawn. In order to distinguish between different graphical elements, we need to add another attribute called *doubleLined* which states if the circle with be drawn with a single or with a double one.
- *Diamond:* describes a simple rhombus. It inherits the x and y attributes from the *Display* super-class which are used to define the position of the center of the rhombus on the screen. It also has an attribute called *height* that defines the distance between the center of the rhombus to any of its vertexes.
- *Triangle:* this basic shape describes a simple equilateral triangle. It inherits the x and y attributes from the *Display* super-class which are used to define the position of the center of mass of the triangle on the screen. Another integer attribute x defines the distance between the center of the triangle to its three vertexes.
- *Group:* is used as a container for other types of displays. A group is displayed as a simple rectangle and is a container of other display elements. The group figure is mainly introduced for representing the *pools* and *lanes* of a business process fragment.
- *Graphical marker:* this is a special kind of display that we propose. It is mainly used for describing the graphical markers used for graphically distinguishing between the different types of events, gateways and data objects. As a graphical marker may define a complex graphical shape, we use *images* for displaying them. For this purpose, a graphical marker contains an attribute called *marker* which is of *resources* type. We have defined seven types of graphical markers, which will all inherit from the *GraphicalMarker* meta-class: *envelope*, *clock*, *error*, *parallel*, *inclusive*, *complex* and *arrow*.

- *Stereotype:* defines a textual annotation that can be added on different graphical elements. A stereotype is defined a special type of *label*. We require the definition of two types of stereotypes: $\langle\langle input \rangle\rangle$ and $\langle\langle output \rangle\rangle$.

Edges are used for connecting the nodes of the graph and transitioning between them. An edge has a single *source* and a single *target*, represented by two reference relations to the *Node* meta-class. Another characteristic of an edge are its *waypoints*. They define the end points of the edge, therefore where it is attached to the source and target nodes. Different *labels* can be added on an edge, in order to bring supplementary information or to express conditions that need to be fulfilled for activating the respective edge. Each **label** consists of some *text* that is associated with the edge. Waypoints are also used to determine how the label position relates to the bounds of the edge geometry. As in the case of nodes, edges have to be graphically represented in a diagram. Therefore, an edge has a graphical *display* which is a *line*. A *line* connects two nodes (points) in the graph. A line is characterized by the two end-points between which it is drawn, which are given be the *waypoints* of the edge. Due to our specific needs, a line can have a variable *thickness*. We distinguish between three types of lines: *continuous* ones, drawn in a single movement between its two end-points; *dashed* lines which are drawn as a succession of several small line segments; *doted* lines. A line can have an *arrowhead*, which is a simple arrow shape that can be attached to the line and graphically represents its directionality. *Arrowheads* may be either *empty* or *filled*.

Once the model of diagrams from Figure 4.13 has been defined, specific diagramming types are described by specialising this model. By refining these concepts to specific model elements, the structure of the graphical concrete syntax of our language is elaborated. We create a new model which relates elements from the abstract syntax to their graphical representation in the concrete syntax. Due to the size of the resulting model and in order to improve the comprehensibility of the diagrams, we decided to split it in two parts, presented in Figure 4.14 and in Figure 4.15.

The meta-model in Figure 4.14 presents the concrete graphical syntax of the part of our language responsible with the modelling of composable business process fragments. The *BP composition language* meta-class, which is the main class of our language meta-model previously described in Figure 4.7, inherits from and is a type of *Diagram*. The graphical syntax of the language is defined as a *Graph*. The language allows to model different *composable business process fragments*. Each of these fragments contains several *CBPF objects*. These objects will be represented in our graph and inherit from either the *node* or the *edge* meta-classes from the general diagram meta-model introduced in Figure 4.13. For example, all the *ConnectingObjects* are types of *edges*. Implicitly, they will be graphically displayed as different types of lines with or without arrowheads. It can be noticed that the *MessageFlow* is displayed as a *DashedLine* with an arrowhead. In order to specify that the associated arrowhead is an empty one, the *IsFilled* attribute of the *Arrowhead* class is set to *false*. As another example, we analyse the *SequenceFlow* meta-class. It is displayed using a *ContinuousLine* with a *filled arrowhead* attached to it. Moreover, as a sequence flow may have a *condition* associated, the line that displays the sequence flow will have a *label* associated to it. The *DataAssociation* relation is defined in a similar manner.

All of the other *CBPF objects* of a composable business process fragment inherit from the *node* meta-class and are accordingly displayed as different types of nodes in the diagram. A *task* is simply displayed by a *rounded-edged rectangle*, while for displaying a *sub-process* we need a *sharp-edged rectangle* with a particular *marker* attached to it. All *gateways*

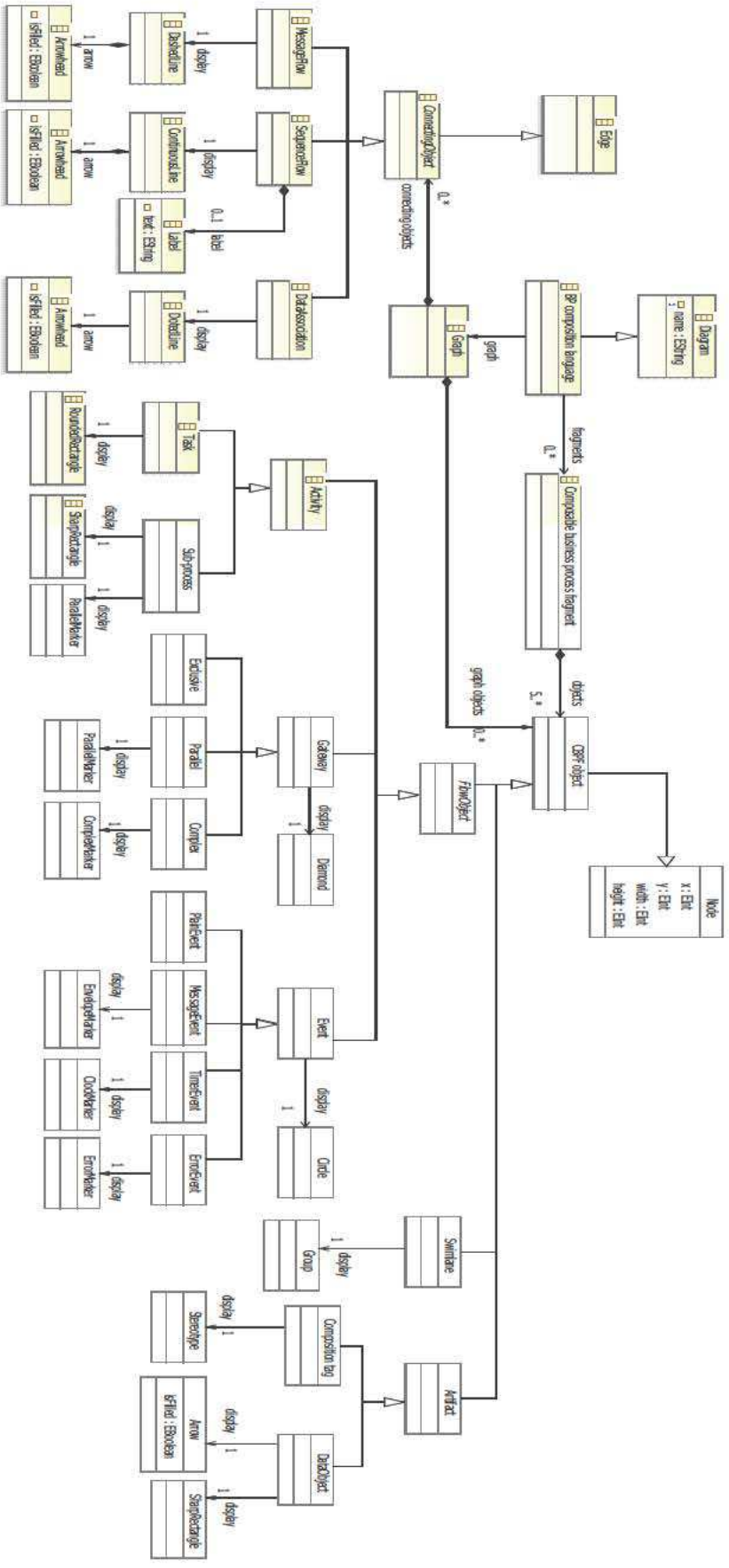


Fig. 4.14: Concrete graphical syntax and relation with abstract syntax - part 1

are displayed using a *diamond*. To differentiate between the different types of gateways, specific *graphical markers* are used, like the *complex marker* for the complex gateway or the *parallel marker* for the parallel gateway. All *events* are graphically displayed using variations of the *circle* shape. As in the case of gateways, graphical markers are used for distinguishing between different events: *envelope marker* for the message event, *error marker* for the error event, *clock marker* for the timer event. Moreover, to distinguish between *start, intermediate* and *end* events, the *doubleLined* and *thickness* attributes of the *Circle* meta-class are used. *Swimlanes* are simply displayed using the *group*, which was specifically introduced in the general diagram meta-model for this purpose. Finally, in the case of *artifacts*, the *composition tags* are displayed using *stereotypes* particularly created for this, while *data objects* are displayed using a *sharp-edged rectangle* and the *arrow marker*.

The meta-model in Figure 4.15 presents the concrete graphical syntax of the part of our language responsible with creating the product derivation specifications. The *graph* used for graphical representation is used for displaying different types of *PDS objects*. *Connectors* inherit from the *edge* meta-class and are graphically displayed using a *continuous line* with a *filled arrowhead* attached to it. The *fragment place-holders* and *operators* inherit from the *node* meta-class. A *fragment place-holder* is simply displayed using a *sharp-edged rectangle*. The composition *operators* are all displayed using *equilateral triangles*. To distinguish between the possible composition operators, each one will also have a specific *stereotype* displayed inside the triangle shape.

Throughout this section we proposed and presented two different manners of defining the graphical concrete syntax of our CBPF language: a straight-forward one which assigns graphical representations to the elements of the abstract syntax meta-model; a second one which defines a separate meta-model for the concrete syntax and proposes a mapping between it and the meta-model of the abstract syntax. In the end, both methods produce the same graphical syntax that will be used for displaying models created with the CBPF language.

In order to facilitate the understanding of the language concepts proposed by the CBPF language and their graphical representation, we present in Figure 4.16 a small example of a business process fragment created with the CBPF language. The goal of this example is to present to the reader how a business process fragment looks like in practice and to show some of the elements it may contain. The example in Figure 4.16 presents a *transportation reservation business process fragment*. It can be easily imagined that such a fragment may be used within a SPL, for example an entire vacation booking SPL. Several of the main elements of a business process fragment are displayed in this example, like: tasks (*select destination*, *book flight*),sub-processes (*book train ticket*), start and end events, exclusive gateway, message events (*phone agent*, *flight info*), error event, all of them connected by sequence and message flow relations. Moreover, we can also see some of the new elements proposed by the CBPF language: composition tags (*input tag* at the start event, *output tag* associated to the end event and the *find alternate transportation* task). For this fragments, the *composition interface* is made up of the start and end event and also of the tagged task. We also wanted to show one of the main characteristics of business process fragments - they represent incomplete information. Thus, it can be noticed that one of the branches of the process ends with a task that has an associated composition tag. This represents that at that particular place, more detailed information will be added by composition with other business process fragments.

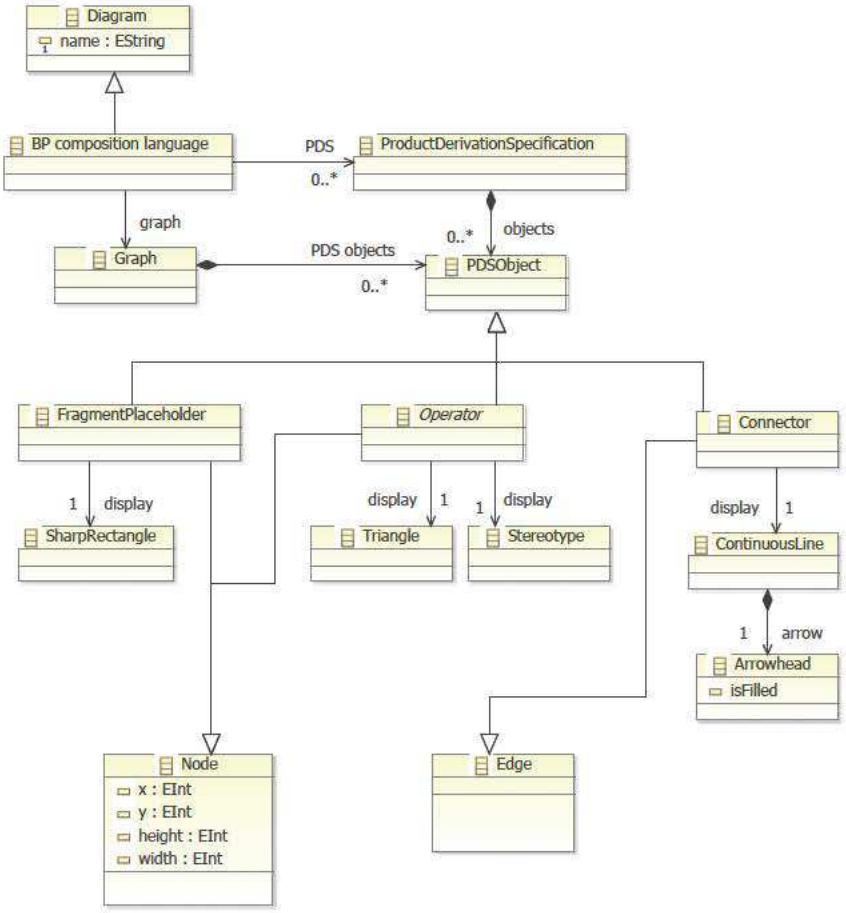


Fig. 4.15: Concrete graphical syntax and relation with abstract syntax - part 2

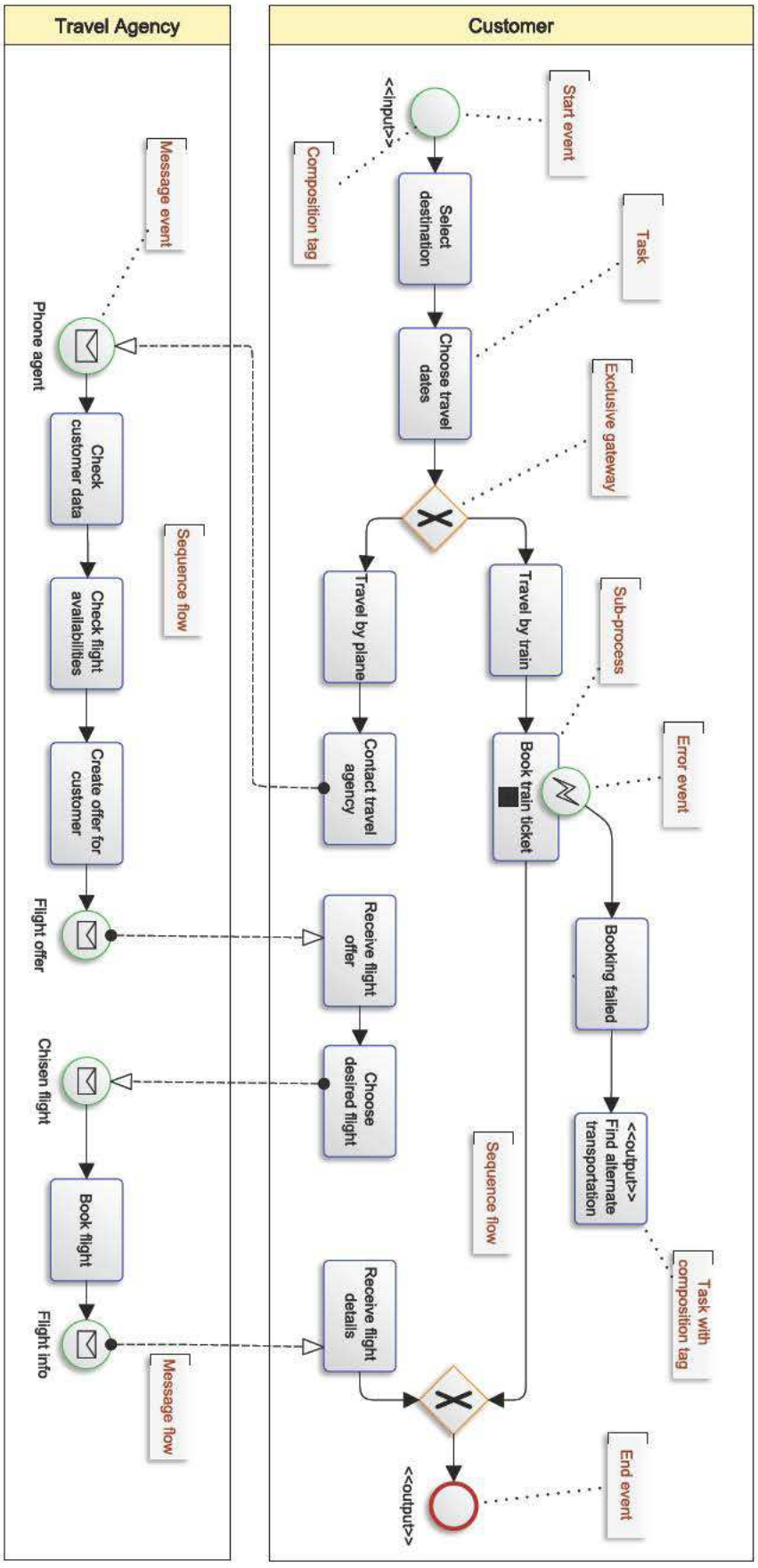


Fig. 4.16: Example of business process fragment created with CBPF : transportation booking

However, the syntax of the language, specified by means of a meta-model, only defines the *structure of the language*. The *semantic properties*, such as conditions over valid models and the behavioural semantics of a model, are not defined. A *semantic description* is included in a language description because the language designer wants to *communicate the understanding of the language to other persons*. Therefore, throughout the following section, we define a translational semantics for the CBPF language. This will complete the language definition.

4.4 Translational semantics

Semantics descriptions of software languages are intended for human comprehension. In general terms, a language semantics describes the meaning of concepts in a language. The semantics can therefore be seen as the *abstract logical space in which models, written in the given language, find their meaning*. Semantics are as important as the structure of the language.

For the description of the language semantics, there are several possible existing approaches, which were presented in Section 2:

- *Operational:* modelling the operational behaviour of language concepts;
- *Axiomatic:* defining a set of properties satisfied by the model in the different steps of its execution (pre- and postconditions);
- *Translational:* translating from concepts in one language into concepts in another language that have a precise semantics.

In this thesis we choose to provide a *translational semantics* for the CBPF language. This kind of semantics is based on two key notions:

- The semantics of a language is defined when the language is translated into another form, called the target language;
- The target language can be defined by a small number of primitive constructs that have a well defined semantics.

The goal of a translational approach is to define the semantics of a language in terms of primitive concepts that have their own well defined semantics. Typically, these primitive concepts will have an operational semantics already defined. The main advantage of this approach is that if the target language can be executed, it is possible to directly obtain an executable semantics for the source language via the translation. Moreover, numerous works use translational semantics mainly to take advantage of the facilities and tools available in the target language (code generators, model-checkers, simulators, visualization tools, etc.). However, a possible disadvantage is that information might be lost during the transformation process. While the end result is a collection of primitives, it will not be obvious how they are related to the original modelling language concepts. There are ways to avoid this: it is possible to maintain information about the mapping between the two models.

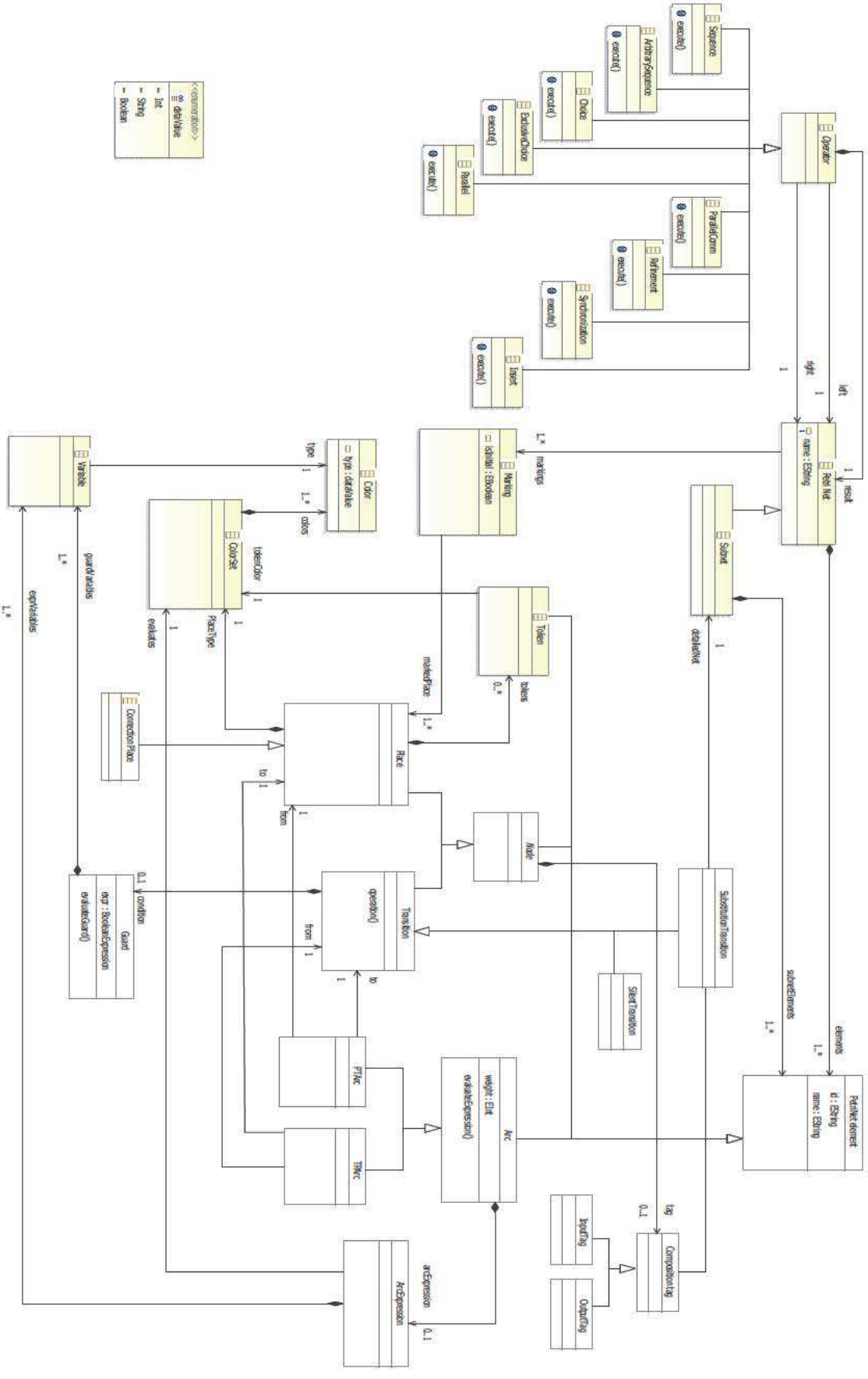


Fig. 4.17: Meta-model of Hierarchical Coloured Petri Nets

We use the translational approach for defining the semantics of the CBPF language. As our language defines a workflow-based notation, we choose hierarchical coloured Petri nets (HCPN) as the target language for the translational semantic definition. HCPN, introduced in section 2.3.4, are a well know and formally defined language. They combine the strengths of ordinary Petri nets with the strengths of a high-level programming language.

4.4.1 Meta-model of Hierarchical Coloured Petri Nets

In order to define the translation between concepts belonging to our language and HCPN concepts, we first need to define a meta-model for the HCPN language. The meta-model we propose is based on the original work of Jensen et al. [Jen94] for defining hierarchical coloured Petri nets, but also on other existing work in the HCPN field from authors like Domokos et al. [DV02], Delatour et al. [DdL03], Hillah et al. [HP10], Weber et al. [WK03b] and the ISO International Standard for High-level Petri Nets [JTC07]. The resulting meta-model is depicted in Figure 4.17.

Petri Net is the main meta-class of the meta-model, characterized by a unique *name*. Every Petri net is composed of several *Petri Net elements*. There are three main types of such elements defined: *nodes*, *arcs* and *tokens*. There are two types of nodes defined: *transitions* and *places*. For defining Petri net composition operators, we define a special type of place called *connection place*. During the execution of the Petri net, each place can contain several *tokens*. Places are characterized by a specific type, called the *colour set*. Such a colour set of the equivalent of a data structure in programming languages, and therefore may contain several *colours*, each one defining a particular type of data. *Tokens* are used for executing a Petri net. A token is characterized by a particular *colour* that defines the type of data it can carry. *Transitions* are another type of *node*. In HCPN, transitions may have a *guard* which defines a particular condition that must be satisfied for the transition to fire. To facilitate the composition of Petri nets, a specific type of transition is introduced: *silent transition* (has no action associated to it). Another category of Petri net elements are *arcs*, which are used for connecting places and transitions. An arc has a single *source* and *target*. Arcs may have associated *arc expressions* that impose certain conditions that may apply for the traversal of the respective arc. The HCPN is also characterized by its *markings*, which are a distribution of tokens over the places of the net. In order to introduce hierarchy, the concept of *subnet* is used. A *subnet* inherits from the *Petri net* meta-class, therefore may contain the same type of elements. Subnets are used for the hierarchical decomposition of HCPNs and to facilitate the modelling and comprehensibility of complex nets. It is possible, in a HCPN, to make a reference to another subnet, through the use of the *substitution transition* concept. They are used for hiding complexity and point towards a detailed subnet which they reference.

In addition to the standard HCPN elements described above, we enrich our meta-model with two more concepts: *composition tags* and *composition operators*. The nodes of a HCPN may have an associated *composition tag*, which explicitly marks that node for further composition. Moreover, HCPNs may be composed between themselves using the classical Petri net composition operators defined in the Petri net literature. Therefore, the *operator* abstract meta-class is introduced in our meta-model. All the operators are binary and take two nets as input and produce a new net as output. There are nine types of composition operators that specialise the abstract super-class by implementing different types of compositions: *sequence, choice, exclusive choice, arbitrary sequence, refinement, synchronization, insertion, parallel* and *parallel with communication*.

4.4.2 Model-to-model transformation from CBPF to HCPN

The semantics of our language is defined in a translational way by a mapping towards Hierarchical Coloured Petri Nets, who have a well-defined formal semantics. The model transformation we propose is described as a series of *mapping rules* or *mapping templates* that translate the elements defined in the abstract syntax of our language into equivalent constructs in HCPN. As our language is much bigger than HCPN in terms of size and number of elements, the mapping will usually not be 1-to-1, but in most cases a language element will be translated into an equivalent set of HCPN elements (a HCPN construct). The mapping templates proposed range from simple 1-to-1 ones, to more complicated.

Before presenting the mapping templates (rules) in detail, we need to introduce some HCPN concepts that are necessary for performing these mappings:

- *Silent transition:* is a particular type of transition (subclass of *Transition* meta-class in Figure 4.17) characterized by the fact that it *has no operation associated to it.* Thus it does not perform any activity once the transition is reached. This type of transition is simply used for mapping purposes, in order to facilitate the definition of the mapping rules. Graphically, silent transitions are represented by black-filled rectangles;

- *Connection place:* is a special type of place (subclass of *Place* meta-class in Figure 4.17) that we introduce. In order to distinguish it from regular places, it is marked with an S symbol and is represented with an interrupted line contour. Connection places are used to represent how the particular element for which we currently propose the mapping connects with other diagram elements, when part of an entire HCPN. Thus, connection places serve only for the mapping itself and will never appear in the actual HCPN diagram. What happens when a business process fragment is transformed into an equivalent HCPN is that every element of the business process fragment is mapped, according to the mapping rules, to an equivalent HCPN construct; then, all the obtained constructs need to be connected together for obtaining the final HCPN; at this moment, the connection places of each HCPN construct will be merged with the corresponding connection places (input or output) of the HCPN construct with which the current construct needs to be related. For example, suppose we have two tasks connected by a sequence flow relation. Based on the mapping templates, each task is translated individually into HCPN as a simple transition with an input and output connection place; to put the two tasks together in the sequence flow relation, the output connection place of task 1 is merged with input connection place of task 2 into a regular HCPN place. Graphically, a connection place is represented as a circle drawn with a dotted line and with has the "S" symbol inside the circle shape.

In the following, we introduce and discuss the proposed mapping rules:

- **Composable business process fragment:** describes at the highest level the business process fragments that we want to create. This concept is mapped to the *Petri Net* concept, which denotes at the highest level a HCPN. The string attribute *title* which denotes the unique name of a composable business process fragment, is mapped onto the string attribute *name* of a Petri net.

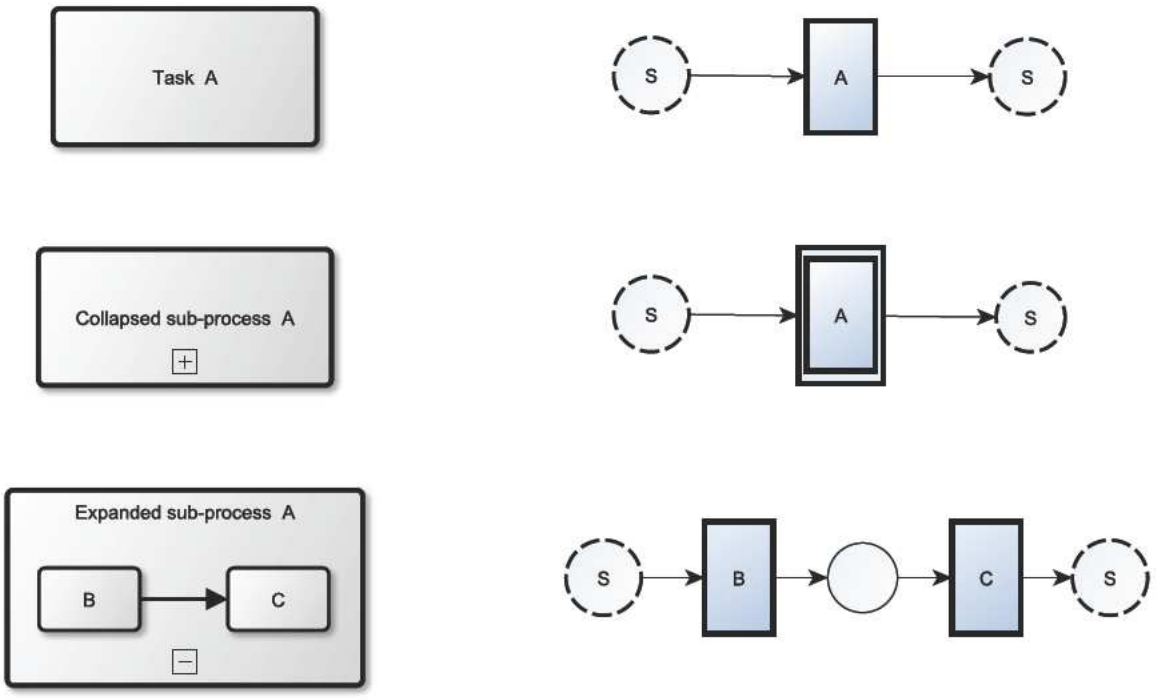


Fig. 4.18: Mapping template from CBPF to HCPN: task and sub-process

Composable business process fragment $\longmapsto$ *Petri Net*, with *Title: string* $\longmapsto$ *PetriNet . Name: string*

- **CBPF object:** a composable business process fragment consists of several elements. A CBPF object provides a high level description for all the elements that can appear in our diagrams. This concept is mapped onto the *Petri Net element* concept in HCPN, whic serves the main purpose for the HCPN language. A *CBPF object* is characterized by two string attributes: *name* and *id*. These attributes are mapped onto the corresponding *name* and *id* string attributes of a *Petri Net element*.
  CBPF object $\longmapsto$ *Petri Net element*, with *name: string* $\longmapsto$ *Petri Net element . name : string* and *id: string* $\longmapsto$ *Petri Net element . id : string*

  To facilitate the understanding of the proposed mapping, Figure 4.18 graphically presents the mapping templates that translate tasks and sub-processes into equivalent HCPN constructs.

- **Task:** defines a basic activity performed in a business process fragment. This concept is mapped onto the following Petri net construct: a *transition* connected to *an input connection place* and *an output connection place*. The transition denotes the processing performed by the task. The input connection place is used to denote the start of the task, while the output connection place denotes the end of the processing proposed y the task. Moreover, these connection places are used for the future connection with other elements, when a task is part of an actual business process fragment. A task also defines an *operation()*, which is mapped onto the corresponding *operation()* which exists for a transition.

- **Collapsed sub-process:** is a compound (non-atomic) type of activity. The *collapsed*

sub-process hides the internal details about the process implementation and thus provides a high-level view of an activity. It is mapped onto the following HCPN construct: a *substitution transition* connected to *an input connection place* and *an output connection place*. The *substitution transition* denotes and is associated to a Petri net subnet. The role of the connection places is to denote the start and end of the sub-process, and as before, they are also used for further connecting a collapsed sub-process with other elements in a business process fragment. We make this mapping as the *substitution transition* concept plays a similar abstraction and hierarchical decomposition role in a HCPN as a sub-process does in a business process fragment.

- **Expanded sub-process:** shows its details within the view of the process fragment in which it is contained. They can be used to flatten a hierarchical process fragment so that all the details can be shown at the same time. We map this concept onto the following HCPN structure: a *subnet* connected to *an input connection place* and *an output connection place*. The role of the connection places is the same as before. In HCPNs, each *substitution transition* is related to a *subnet* providing a more detailed description than the transition itself. It is clear thus that the role of a subnet in a HCPN is very similar to that of a *collapsed sub-process* for business process fragments. Moreover, the existing logical relation between collapsed and expanded sub-processes is kept by mapping them to substitution transitions and subnets, which hold a similar relation between them.

- **Parallel gateway:** this type of gateway either splits one incoming sequence flow into multiple outgoing parallel paths of execution or merges and synchronizes multiple incoming flows. There are two possible types of this gateway, each with its own mapping:

  - **Splitting:** we map this concept onto the following HCPN structure: an *input connection place* followed by a *silent transition*; from the silent transition, there are two (or more) output arcs, each one connecting the silent transition with an *output connection place*. We require such a complicated Petri net structure as we want to keep the original semantics of the parallel gateway and conserve the behaviour. The connection places are used as before for the future connection with other elements. The silent transition, which does not execute any specific activity, is used for splitting the outgoing execution paths and for preserving the original semantics of the parallel splitting gateway in Petri net.
  - **Merging:** we map this concept onto the following HCPN structure: two (or more) *input connection places* are connected by arcs with one *silent transition*; this transition is then connected through an arc to an *output connection place*. The idea of the mapping is very similar with the one for the splitting parallel gateway, adapted to the particularities of the merging gateway. The role of the silent transition is to merge the incoming sequence flows and preserve the original semantics of the parallel merging gateway in Petri net.

- **Exclusive gateway:** this type of gateway denotes a decision point in the flow of execution. For the splitting gateway, the incoming flow is split into multiple outgoing paths, from which exactly one can be taken. For the merging gateway, only one of the incoming flows may lead to the output flow:

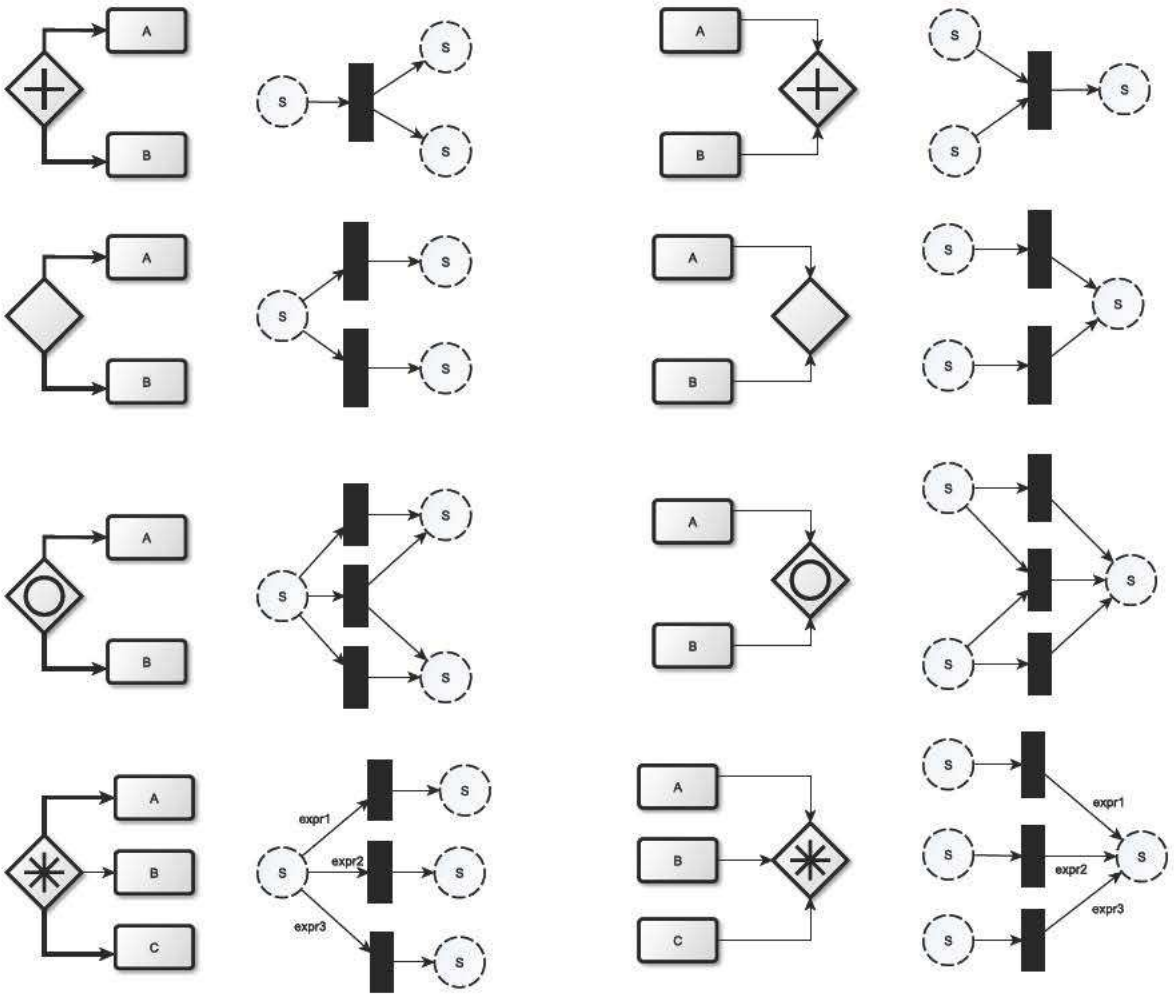


Fig. 4.19: Mapping template from CBPF to HCPN: gateways

- **Splitting:** we map this concept onto the following HCPN structure: an *input connection place* has two (or more) outgoing arcs, each leading to a *silent transition*. Each such silent transition is then connected to an *output connection place*. The number of silent transitions is given by the number of outgoing sequence flows of the exclusive gateway. During execution, once a token has passed from the input connection place through one of the silent transitions, none of the other paths can be accessed any more. This behaviour coincides to the one described by the exclusive gateway.
- **Merging:** we map this concept onto the following HCPN structure: two (or more) *input connection places* are connected each one to *silent transitions*. All these silent transitions are then connected through their outgoing arcs to a single *output connection place*. There are as many silent transitions as input sequence flows for the exclusive merging gateway.

- **Inclusive gateway:** this type of gateway denotes a decision point in the flow of execution. For the splitting gateway, the incoming flow is split into multiple outgoing

paths, from which one or more can be taken. For the merging gateway, one or more of the incoming flows may lead to the output flow:

  - **Splitting:** the mapping of this gateway is inspired by the one previously presented for the exclusive splitting gateway. Therefore, we start from the Petri net structure obtained for the exclusive gateway and enrich it. The *input connection place* contains an extra outgoing arc leading to another silent transition. This silent transition will have two outgoing arcs, each one leading to the already existing *output connection places*.
  - **Merging:** the mapping of this gateway is inspired by the one previously presented for the exclusive merging gateway. Therefore, we start from the Petri net structure obtained for the exclusive gateway and enrich it. Each one of the *input connection places* has an extra outgoing arc connected to a new*silent transition*. This silent transition is then connected through an outgoing arc to the *output connection place*.

- **Complex gateway:** this type of gateway denotes a complex decision point in the flow of execution, which cannot be easily expressed with the previously described gateways. A complex gateway contains a boolean condition attribute that specifies an expression that determines which and how many of the sequence flows will be chosen for the process fragment to continue. To express this condition for our mapping, we make use of the *arc expression* concept that exists in HCPN.

  - **Splitting:** we propose the following mapping: one *input connection place* is connected by outgoing arcs to three (or more) *silent transitions*. Each of these outgoing arcs has an associated *arc expression*. Each of the silent transitions is then connected by a simple arc to its own *output connection place*.
  - **Merging:** we propose the following mapping: three (or more) *input connection places* are each connected through simple arcs to *silent transitions*. Each such silent transition is is then connected by an outgoing arc to a unique *output connection place*. Each of these arcs has an associated *arc expression*. Therefore, the condition of the complex gateway is replaced by having arc expressions on each of the arcs connecting the silent transitions to the unique output connection place.

  For a better understanding, Figure 4.19 graphically presents the mapping templates that translate all the types of gateways into equivalent HCPN constructs.

- **Start event:** indicates where a particular business process fragment will start. Depending on the type of trigger that a start event might have, we propose to map this element to the following HCPN structures:

  - *Plain start event:* is mapped onto a *place* with no input arcs and one outgoing arc. Moreover, this place will hold the initial marking of the Petri net;
  - *Message start event:* indicates that a message arrives from a participant and triggers the start of the business process fragment. We propose to map this element to the following HCPN structure: a *place* with no input arcs and one outgoing arc leading to a *message transition* and then an *output connection place*. The *message transition* is discussed a bit later;

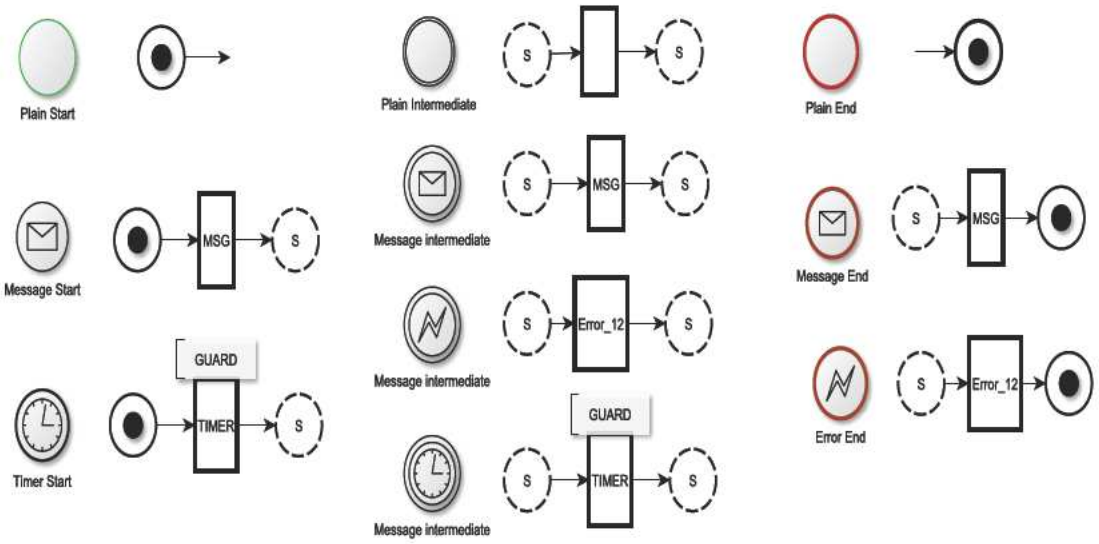


Fig. 4.20: Mapping template from CBPF to HCPN: events

– *Timer start event:* denotes that the elapse of a certain time interval triggers the start of the process. We propose to map this element to the following HCPN structure: a *place* with no input arcs and one outgoing arc leading to a *timer transition* and then an *output connection place*. The *message transition* is described a bit later;

- **Intermediate event:** denotes that something happens inside the flow of the business process fragment. Depending on the type of trigger that a start event might have, we propose to map this element to the following HCPN structures:

  – *Plain intermediate event:* is mapped onto an *input connection place* followed by a simple *transition* connected then to an *output connection place*. The role of the input connection place is to denote the start of the event, while the output connection place denote the end of the event. The connection places are also used for connectivity reasons when the event belongs to a HCPN;

  – *Message intermediate event:* we propose to map this element to the following HCPN structure: an *input connection place* related through an arc to a *message transition* and then an *output connection place*. The *message transition* is a special type of transition used to simulate the sending or receiving of a message. It has the following characteristics: the string attribute *name* is set to have the value *"Message"*; the generic *operation()* method is defined to be either *sendMessage(msg: String)* or *receiveMessage(msg: String)*;

  – *Timer intermediate event:* may denote that a specific time-date or a specific time cycle can be set that will trigger the start of the process fragment or act as a delay mechanism. We propose to map this element to the following HCPN structure: an *input connection place* related through an arc to a *timer transition* and then an *output connection place*. The *message transition* is a special type of transition used for simulating time intervals. It has the following characteristics: the string attribute *name* is set to *"Timer"*; a *guard* acting as a condition to to

show when the specified time interval has passed or not; the guard expression is defined based on the original *TimeDate* of the timer event;

- *Error intermediate event:* signals an error in the functioning of the process fragments and disrupt the normal flow of activities. We propose to map this element to the following HCPN structure: an *input connection place* related through an arc to a *error transition* and then an *output connection place*. The *error transition* is a special type of transition used for defining the occurrence of errors. It has the following characteristics: the string attribute *name* is set to *"Error"*; the string that determines the error code is concatenated to the name attribute, resulting in a transition name of the type: *"Error*$_e$*rrorCode*

- **End event:** indicates where a process will finish. Depending on the type of trigger that a start event might have, we propose to map this element to the following HCPN structures:

  - *Plain end event:* denotes the normal termination of a business process fragment. This element is mapped onto a simple *place* that has no outgoing transitions, which will also hold the final marking of the Petri net;
  - *Message end event:* denotes that the sending or receival of a message causes the business process fragment to stop. We propose to map this element onto an *input connection place* related through an outgoing arc to a *message transition* which is then connected to a simple *place* that has no outgoing transitions;
  - *Error end event:* denotes the fact that the occurrence of an error cause the business process fragment to end. This element is mapped onto an *input connection place* related through an outgoing arc to an *error transition* which is then connected to a simple *place* that has no outgoing transitions.

  For a better understanding, Figure 4.20 graphically presents the mapping templates that translate all the types of events into equivalent HCPN constructs.

- **Sequence flow:** shows the order in which flow objects are executed in a business process fragment. A sequence flow relation is characterized by one source and one target. There are three possible types of sequence flows:

  - *Normal:* defines the exact order of execution of flow objects in a process fragment. We propose to map this element onto the *arc* concept in HCPN. The obtained arc will have *no arc expression associated to it*. The boolean attribute *condition* is mapped onto the *arc expression* concept in HCPN.
  - *Conditional:* has a conditional expression attribute that must be evaluated before the sequence flow can be traversed. This concept is mapped onto *an arc with an associate arc expression* in HCPN. The arc expression servers a similar role as the conditional expression, restricting certain tokens to traverse the respective arc.
  - *Exception handling:* this particular type of sequence flow occurs outside the normal flow of the process and is based upon an event (intermediate error) that occurs during the execution of the process fragment. Error events are usually attached to tasks of a business process fragment. The modeller is creating an event context to interrupt the activity and redirect the flow through the intermediate error event. We map this situation into the following HCPN construct:

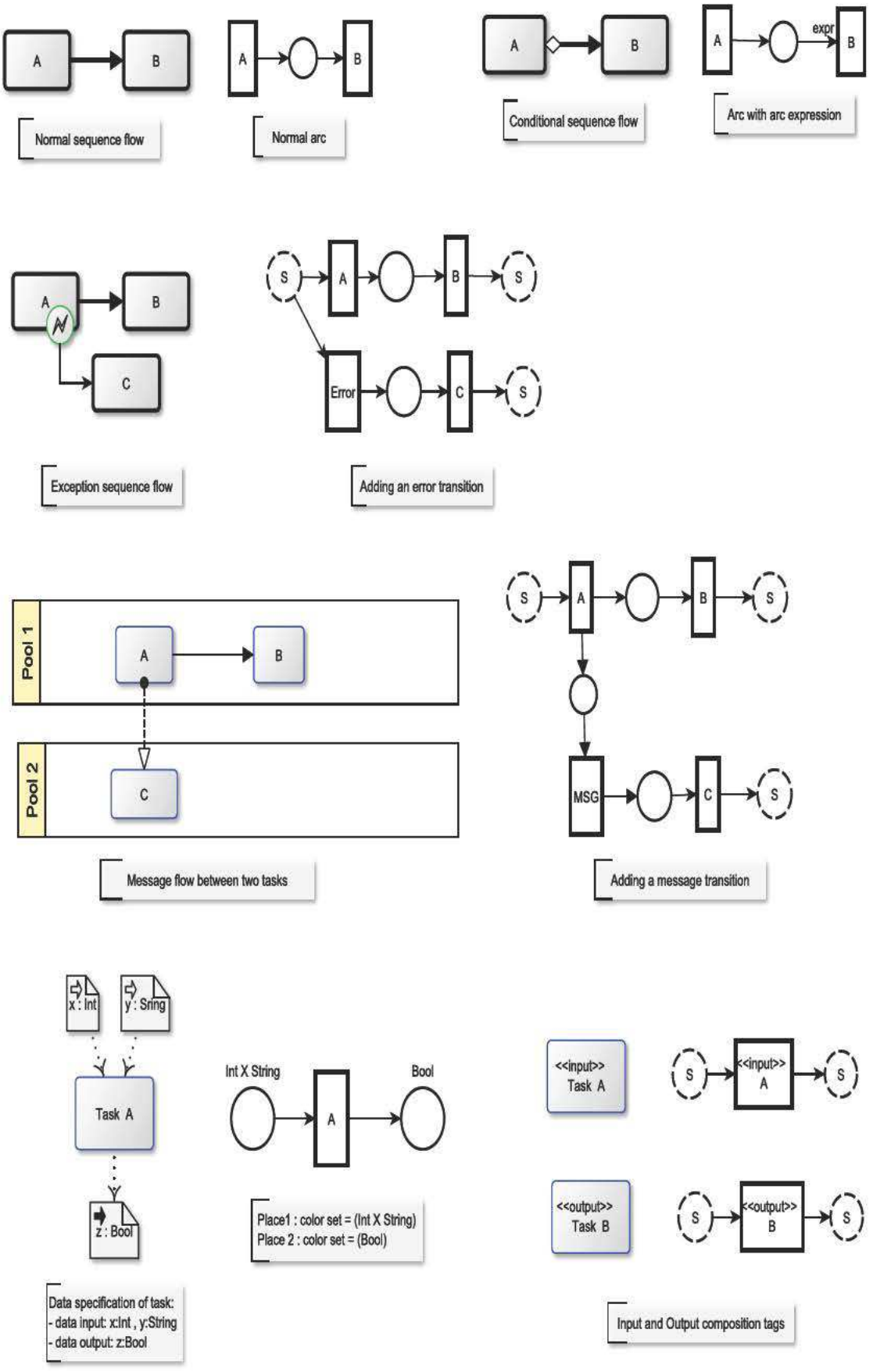


Fig. 4.21: Mapping template from CBPF to HCPN: sequence flow, message flow, data association, data objects and composition tags

an exception flow attached at the input connection place of the affected task (to which the error event is attached). We made the explicit choice to attach the exception flow before the task, although it is also possible to attach it after after the task.

- **Message flow:** shows the flow of messages (message exchange) between two entities. Message flow must connect two pools, either to the pools themselves or to some flow objects within those pools, but it cannon connect two objects belonging to the same pool. The message flow is interpreted as a message exchange between the two flow objects belonging to different pools. Therefore, it is translated into a HCPN *message transition* that will be placed between the HCPN constructs corresponding to the translated flow objects.

- **Swimlanes:** are used to help partition and organize the activities of a business process fragment. A swimlane is thus a way to contain and group the flow objects that are performed by a certain participant. There are two types of swimlanes: pools and lanes. As this concept is used for the grouping of elements in a business process fragment, we consider that it should not be mapped onto any HCPN construct. A mapping would however be possible, but it would only overburden the resulting Petri net and make the resulting diagram difficult to understand and possibly illegible. Moreover, such a mapping would not bring any crucial or important information in the resulting HCPN. Therefore, neither pools nor lanes are mapped onto HCPN constructs.

- **Data objects:** are used for modelling data and data flow in a business process fragment. Activities often required data in order to execute. In addition they may produce data during or as a result of execution. Therefore, we propose two types of data objects: *data input* and *data output*. We propose to map such *data objects* onto the concept of *color* from HCPN. This mapping is motivated by the fact that in a HCPN every token has a value also called its colour, which is thus used in HCPN to represent the data type of a token. This which is very similar with the role of data objects in business process fragments.

  A *data object* has also an associated *data type* (can be int, string or bool). The data type of a data object is mapped onto the *type* attribute of a *color*. This attribute defines three different possible data values: int, string, boolean.

- **Data specification:** regroups all the data dependencies of a task. It contains one *InputSet* of data that is processed by the task, and one *OutputSet* of data that is produced by that task. This concept is mapped onto the *color set* notion in HCPN. A color set is a sort of data record that regroups all the data types that a place may contain. As a data specification contains all the data inputs and outputs of a task, similarly a color set regroups all the colors that a HCPN place may have.

- **Data association:** is a specific type of connecting object that allows to connect a task to its data inputs and outputs. As the data object concept was mapped before onto the notion of color in HCPN, a data association does not need to be mapped by itself to any HCPN notion. It is actually the data association together with the corresponding data object that will be mapped onto a specific color in HCPN.

- **Composition tags:** serve to render business process fragments "composable". A composition tag is simply a text annotation in form of a stereotype that can appear

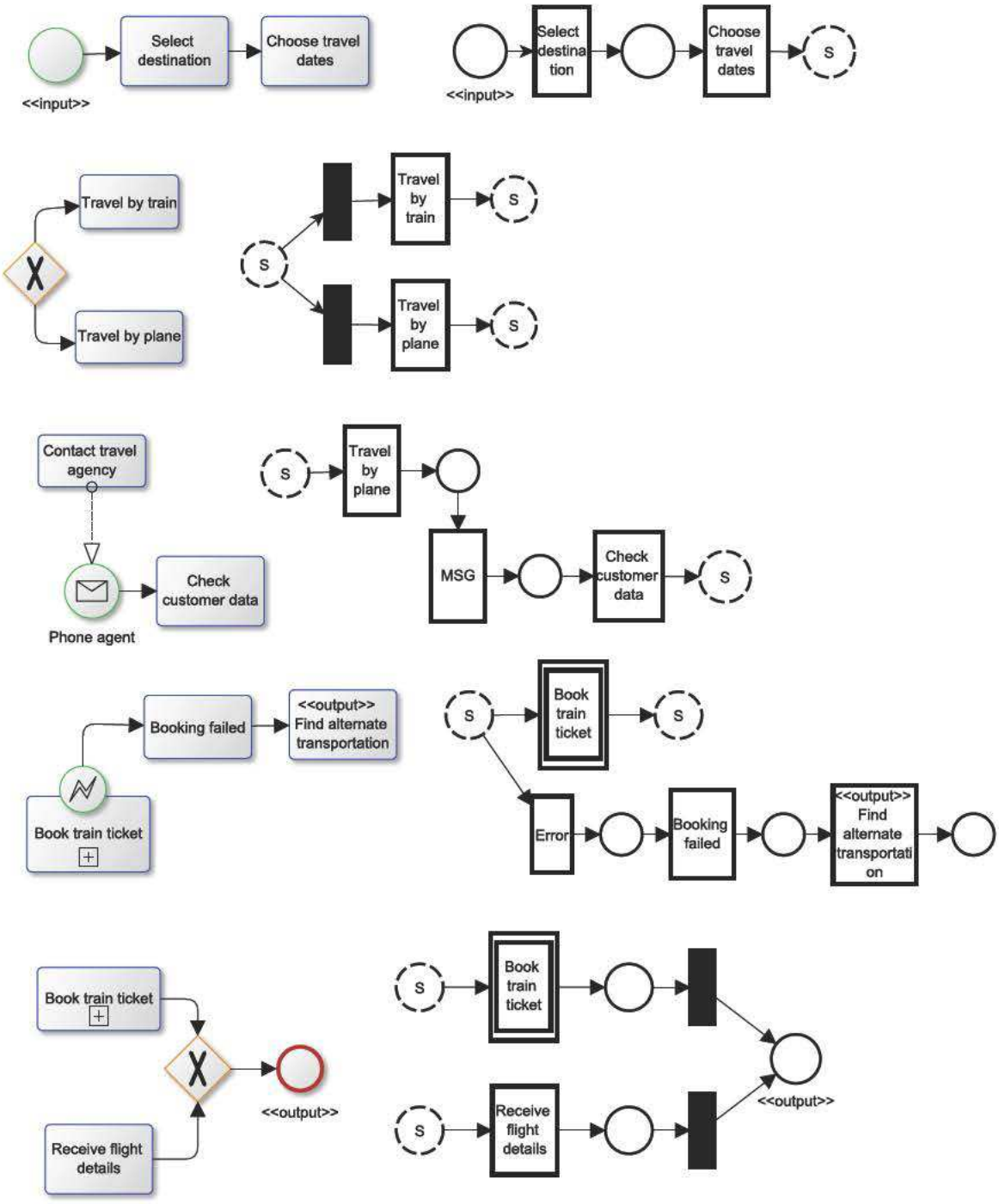


Fig. 4.22: Mapping transportation reservation fragment to HCPN: Step 1

on elements of a business process fragment. A composition tag identifies an exact place in a business process fragment where the fragment can be composed with other ones. This concept is mapped onto the corresponding notion of *composition tag* from Petri net, introduced in our HCPN meta-model from Figure 4.17. Therefore, the *input* and *output composition tags* are mapped onto their HCPN equivalent.

For a better understanding, Figure 4.21 graphically presents the mapping templates that translate sequence flow, message flows, data association, data objects and composition tags into equivalent HCPN constructs.

- **Composition operators:** earlier in this chapter we defined a set of binary composition operators which take two business process fragments as input and produce a single process fragment as output of the composition. When these composition operators were introduced, their semantics was informally described in terms of token passing, in order to facilitate their understanding. In order to formally define their semantics, these operators are mapped onto equivalent composition operators that have been defined for Petri nets. The HCPN composition operators onto which we perform the mapping are classical operators defined for composing high level Petri nets, which can be found in the Petri net research literature. These composition operators are present in the HCPN meta-model from Figure 4.17.

  Therefore, the *operator* concept from CBPF is mapped onto the equivalent *operator* concept from HCPN. Similarly, we have the following mappings:

  - *Sequence:* is mapped onto the sequential composition operator defined for HCPN;
  - *Arbitrary sequence:* is mapped onto the unordered (arbitrary) sequence composition operator defined for Petri nets;
  - *Choice:* is mapped onto the operator with the same name defined for Petri nets;
  - *Exclusive choice:* mapped onto the exclusive choice composition operator for HCPN;
  - *Parallel communication:* is mapped onto the parallel with communication composition operators defined for Petri nets;
  - *Refinement:* is mapped onto the place/transition refinement composition operator for Petri nets;
  - *Synchronization:* is mapped onto the place fusion / transition synchronization operator fro Petri nets;
  - *Insert:* is mapped onto the insert fragment operator defined for Petri nets.

  A review and detailed presentation of the composition operators defined for HCPN is available in Annex 1.

In order to facilitate the understanding of the mapping presented throughout this section, we will explain it on the example previously introduced at the end of Section 4.3 in Figure 4.16. The example described a transportation reservation business process fragment. The mapping towards HCPN is a two.step process. First, based on the mapping templates proposed, all the elements of the business process fragment are transformed into equivalent HCPN constructs. This step of the process is exemplified in Figure 4.22, in which we take several elements (or element groups) from the business process fragment and, applying the mapping templates, transform them into their equivalent HCPN constructs. Then,

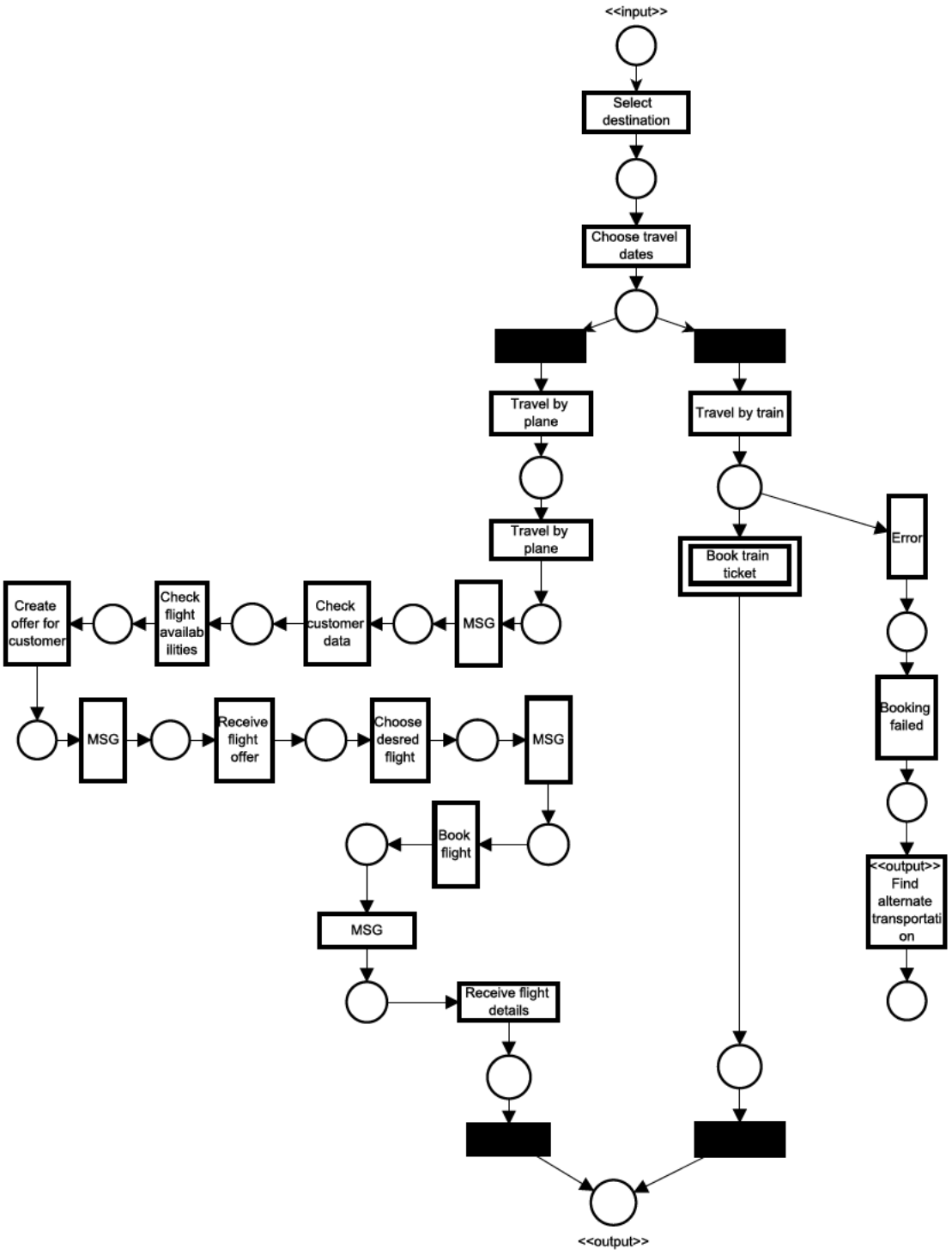


Fig. 4.23: Mapping transportation reservation fragment to HCPN: Step 2

in a second step, all the HCPN constructs obtained in the previous step, are assembled together in order to obtain the final resulting HCPN, whicg defines the transformation of the entire business process fragment into HCPN. For our example, the resulting HCPN model is presented in Figure 4.23.

5. VERIFICATION OF BUSINESS PROCESS FRAGMENT CORRECTNESS

Abstract

Throughout this chapter we propose several types of verifications that can be applied to business process fragments in order to determine their "correctness". Business process fragment verification is a key step of the SPL methodology proposed in Chapter 3. It is highly desirable to verify business process fragments created at analysis and design time. We want to ensure that the business process fragments created with the CBPF language during the domain engineering phase are correct. We start by defining the notion of "correctness" for business process fragments in Section 5.1 as the summation of two other properties: structural correctness and behavioural correctness. In Section 5.2 we present the structural verification of a business process fragment by defining a set of adequate fragment consistency rules that should be valid for every business process fragment that can be created with the CBPF language. These well-formedness rules are defined using OCL directly on the CBPF meta-model. We also want to perform checks related to the dynamic behaviour of business process fragments. Thuse, the verification of behavioural correctness of business process fragments is presented in Section 5.3. These verifications are done by first transforming the business process fragment into an equivalent HCPN with the help of the model-to-model transformation that we propose. Once this is done, we can take advantage of the large array of analysis and verification techniques and tools available for Petri nets. The behavioural properties that should be verified for a business process fragment are separated into two major classes. Generic ones which specify general dynamic properties that any business process fragment should fulfil. As business process fragments are created to describe a high level functionality or feature, there will exist certain dynamic properties that are specific to each individual fragment and therefore cannot be verified in general. Therefore we define a set of fragment specific properties and propose property templates that can be adapted and used by the product line engineer to check them.

As business processes have become more complex, the probability of making errors in their design has increased. Errors in business processes can cause big financial losses, therefore the need for identifying and correcting the errors has become critical. In many cases, business process analysis is often performed by walk-through only. Simulations can also be used for model validation and testing, but verification is needed to guarantee behavioural properties. Real-life business processes are too large and complex to be verified manually, and automated support is therefore essential.

Business process fragment verification is a key step of the SPL methodology proposed in Chapter 3. We want to ensure that the business process fragments created with the CBPF language during the domain engineering phase and the used during the application engineering phase for creating behavioural models of the derived SPL products are *correct*. It is highly desirable to verify business process fragments created at analysis and design time. In a late state of the system development process the cost to repair incorrect business processes are extremely high. Therefore, it is reasonable to identify errors at design time. Moreover, the correctness of a business process specification is critical for the automation of business processes. For this reason, errors in the specification should be detected and corrected as early as possible.

Throughout this chapter we discuss the notion of *correctness* and how it applies to business process fragments. We will see that guaranteeing the *correctness* of a business process fragment implies ensuring that two properties are verified: *structural correctness* and *behavioural correctness*. The particularities of each of these specific verifications are separately discussed in detail throughout this chapter.

5.1 Notion of *"correctness"* for business process fragments

The notion of *"correct business process"* has a wide understanding in business process research. In general, by correctness properties, people usually refer to the different kinds of *soundness properties* first introduced in 2000 in the work of Wil van der Aalst on workflow verification [vdA00]. The notion of soundness was later on also extended to business processes.

The verification of business process fragment *correctness* is essential for ensuring an unambiguous description of the processes. In this thesis, we analyse the notion of business process correctness from a different perspective and try to adapt it to the particularities of business process fragments. Therefore, we define the notion of *correctness* for business process fragments as the summation of two other properties: *structural correctness* and *behavioural correctness*. We introduce each of these two types of properties in the following:

- *Structural correctness:* mainly focuses on avoiding errors at the structural level of the business process fragments. For the CBPF language and thus business process fragments, this property deals with:
  - the correspondence between the model and the language in which the model is written (in out case CBPF);
  - the alignment between a business process fragment model created using CBPF and a set of structural properties that any model of the same type must respect.

  Structural properties mainly refer to the type of elements that may appear in a business process fragment and the various control flow relations that exist between them. More precisely, to ensure the structural correctness of a business process fragments, we first need to define a *set of adequate fragment consistency rules* that should be valid for every business process fragment that can be created with the CBPF language. As the CBPF language was defined following a model-drive approach and its abstract syntax specified as a meta-model, the consistency rules that will ensure the

structural correctness of business process fragments will be defined directly on the business process fragment meta-model. Implicitly, every business process fragment created that is conform to the CBPF meta-model will be ensured to satisfy these consistency rules. We thus propose a set of consistency rules that try to ensure the structural well-formedness of business process fragments, defined using the Object Constraint Language (OCL). Two types of rules are proposed:

  - *Based on OMG BPMN specification:* as the CBPF language shares a large set of elements with the BPMN language, we consider important to keep in our language the consistency criteria defined by the BPMN standard which are relevant for CBPF. However, the BPMN documentation does not define well-formedness rules/criteria for business processes in an explicit and concise manner. This information appears only textually in the BPMN standard [OMG11] as part of the description and presentation of the different BPMN language elements. Therefore, we needed to extract these rules and express them formally using OCL;
  - *Business process fragment specific constraints:* CBPF specific consistency rules are introduced, also expressed using OCL. These rules mainly refer to two aspects: the fact that business process fragments model partial information and the existence of composition interfaces for business process fragments.

- *Behavioural correctness:* mainly concerns the control-flow of the business process fragments. At this level, we want to perform checks related to the dynamic behaviour of business the process fragments. The concept is defined based on the original definition of *process soundness* proposed by van der Aalst for workflow nets [vdA00]. The soundness criterion and its derivatives are typically used to check whether proper completion of business process is possible or even guaranteed. We extend this notion and adapt it for business process fragments. In our case, behavioural correctness ensures that a business process fragment does not exhibit any erroneous or unwanted behaviours. As the behaviour of a business process fragments is defined by its execution traces, the verification of behavioural correctness is also performed on these traces. We propose to verify two kinds of behavioural properties for business process fragments:

  - *Generic:* the goal here is to specify general dynamic properties that any business process fragment should fulfil. We verify high-level properties like the reachability of end events or of composition interfaces, ensure that the process is deadlock-free and has no dead tasks or that the process fragment can properly finish;
  - *Process fragment specific:* certain properties cannot be verified in general and are different form one business process fragment to another and depend on the specific context in which that fragment is used. We want to offer the product line engineer the possibility to define and verify such business process fragment specific properties. These properties concern only the business process fragments themselves and are not related to the different composition operators that might be used for creating new fragments following a compositional approach. Analysing how these properties are conserved and impacted by the different composition operators is planned as future work and some ideas regarding this subject are presented in Chapter 7.

The notion of correctness is an essential one because we want to ensure that all the business process fragments that can be created with the CBPF language respect some structural well-formedness properties, but that also, from a behavioural perspective, our business process fragments poses several dynamic properties that we consider important.In the next sections, we discuss in detail how the structural and the behavioural verifications are performed.

5.2 Verification of structural correctness of business process fragments

The structural verification of a business process fragment requires the definition of a *set of adequate fragment consistency rules* that should be valid for every business process fragment that can be created with the CBPF language. In model driven engineering, a meta-model is typically not refined enough to provide all the relevant aspects of a specification. There is a need to describe additional constraints about the objects in the model. Such constraints are often described in natural language. However, this will always result in ambiguities. In order to write unambiguous constraints, formal languages have been developed. The disadvantage of traditional formal languages is that they are only usable to persons with a strong mathematical background, but difficult to understand and use by the average business or system modellers.

The Object Constraint Language (OCL) [Gro06b] was developed to fill this gap. It is a formal language that remains easy to read and write. It provides a formal language for specifying constraints which can supplement the models and meta-models created following MDE principles. The language has a precise syntax that enables the construction of unambiguous statements. It is used for precisely defining the well-formedness rules for UML and further OMG-related meta-models.

The core concept defined by OCL is that of *constraint*. Warmer and Kleppe [WK03a] define a constraint as *"a restriction on one or more values of (part of) an object-oriented model or system"*. A constraint is formulated on the level of classes or meta-classes, but its semantics is applied on the level of objects. There are three basic types of constraints that can be defined using OCL:

- *Invariants:* define constraints that must always be met by all instances of the (meta)class. These constraints should be true for an object during its complete lifetime;
- *Pre-conditions:* refer to the operations of a model and specify constraints that must always be true *before* the execution of the operation. The meaning of a precondition is that it has to be valid in the initial state of an operation, otherwise the operation should not be executed;
- *Post-conditions:* similarly to pre-conditions, they also refer to the operations of a model and specify constraints that must always be true *after* the execution of an operation/method. They define the way the actual effect of an operation is described in OCL.

We want to establish and ensure that no ill-formed business process fragment models can be produced given the language meta-model. In other words, we want to be sure of the

well-formedness of all the model instances that can be created with the proposed language. It is imperative to check the correspondence between the models and the language in which the models are written. To be sure that the business process fragments that can be created with the proposed language, we will check the alignment between the created models and a set of structural properties that any model of the same type must respect. The approach followed allows us to express a set of desired well-formedness constraints in the Object-Constraint Language with respect to the meta-model of the business process fragment modelling language.

Using OCL well-formedness constraints, it is possible to express different properties and characteristics of business process fragments which cannot be expressed directly with the meta-model. These constraints, defined directly on the language meta-model, facilitate a more refined specification of business process fragments and restricts the set of structurally valid process fragments. Moreover, the advantage of using OCL resides in the fact that the constraints are defined only once, on the language meta-model, and apply to all models created with that meta-model. Using this approach, a model is well-formed if and only if it conforms to the meta-model, i.e., it satisfies the multiplicities and the OCL constraints defined on the meta-model.

The BPMN standard document defines, in an informal manner, using a natural language description, several structural properties that should be verified by all BPMN models that conform to the standard. Based on this document, we adapt some of those constraints to the specificities of our business process fragment modelling and composition language and create a set of well-formedness constraints applicable to business process fragments, which we specify using OCL on the language meta-model. Another set of OCL constraints are well-formedness rules specific to our language, which we create from scratch. All the well-formedness rules are specified using OCL and described in the following. The constraints are first described using natural language, then the corresponding OCL rule is stated:

- There is only a unique instance for a business process fragment:

  context $BPMN_process_fragment$
  inv $uniqueInstance : self.allInstances() \longrightarrow forAll(p_1, p_2 | p_1 = p_2)$

- A business process fragment must respect the following minimal requirements:

  - are single entry workflows - there is exactly one start event e_s (fragment entry) for the process fragment:

    context $BPMN_process_fragment$
    inv $exactlyOneStart : self.fragment_objects \longrightarrow FlowObject \longrightarrow Event \longrightarrow select(e|e.type = EventType :: start) \longrightarrow size() = 1$

  - are multiple exit workflows, i.e. there is at least one end event e_s (fragment exit) for the process. It is allowed to have more than one end event:

    context $BPMN_process_fragment$
    inv $multipleEnds : self.fragment_objects \longrightarrow FlowObject \longrightarrow Event \longrightarrow select(e|e.type = EventType :: end) \longrightarrow size() \geq 1$

  - have at least one activity:

    context $BPMN_process_fragment$
    inv $atLeastOneActivity : self.fragment_objects \longrightarrow FlowObject \longrightarrow Activity \longrightarrow size() \geq 1$

  - have at least two connecting objects (sequence flow relations) between e_s - a and a - e_e):
    context $BPMN_process_fragment$
    inv $atLeastTwoSeqFlows : self.fragment_objects \longrightarrow ConnectingObject \longrightarrow SequenceFlow \longrightarrow size() \geq 2$

- All start events have zero incoming and one outgoing sequence flow relations:

  context $Event$
  inv : $self.allInstances \longrightarrow forAll(e|e.type = EventType :: start\ implies\ self.FlowObject.sourceFlow \longrightarrow size() = 0)$
  inv : $self.allInstances \longrightarrow forAll(e|e.type = EventType :: start\ implies\ self.FlowObject.targetFlow \longrightarrow size() = 1)$

- All end events have zero outgoing and one incoming sequence flow relations:

  context $Event$
  inv : $self.allInstances \longrightarrow forAll(e|e.type = EventType :: end\ implies\ self.FlowObject.sourceFlow \longrightarrow size() = 1)$
  inv : $self.allInstances \longrightarrow forAll(e|e.type = EventType :: end\ implies\ self.FlowObject.targetFlow \longrightarrow size() = 0)$

- All the activities and intermediate events of a process fragment have exactly one input and one output sequence flow relations:

  context $BPMN_process_fragment$
  inv : $self.fragment_objects.FlowObject \longrightarrow select(a|a.oclIsTypeOf(Activity)) \longrightarrow forAll($
  $self.sourceFlow \longrightarrow size() = 1)$

  context $BPMN_process_fragment$
  inv : $self.fragment_objects.FlowObject \longrightarrow select(e|e.oclIsTypeOf(Event)) \longrightarrow forAll($
  $e|e.type = EventType :: intermediate\ implies\ e.sourceFlow \longrightarrow size() = 1)$

- All splitting gateways have an input degree of 1 and and output degree of at least 2:

  context $Gateway$
  inv : $self.allInstances \longrightarrow forAll(g|g.type = GatewayType :: forking\ implies\ self.FlowObject.sourceFlow \longrightarrow size() = 1)$

  inv : $self.allInstances \longrightarrow forAll(g|g.type = GatewayType :: merging\ implies\ self.FlowObject.targetFlow \longrightarrow size() \geq 2)$

- All merging gateways have an input degree of at least 2 and and output degree of 1:

  context $Gateway$
  inv : $self.allInstances \longrightarrow forAll(g|g.type = GatewayType :: merging\ implies\ self.FlowObject.sourceFlow \longrightarrow size() \geq 2)$

  inv : $self.allInstances \longrightarrow forAll(g \mid g.type = GatewayType :: merging\ implies\ self.FlowObject.targetFlow \longrightarrow size() = 1)$

- All error events have no incoming and one outgoing flow relation:

  context $Event$
  inv : $self.allInstances \longrightarrow forAll(e|e.oclIsTypeOf(ErrorEvent)\ implies\ self.FlowObject.sourceFlow \longrightarrow size() = 0)$

inv : $self.allInstances \longrightarrow forAll(e|e.oclIsTypeOf(ErrorEvent)\ implies\ self.FlowObject.targetFlow \longrightarrow size() = 1)$

- Every error event is attached to an activity (either a sub-process or a task):

  context *Event*
  inv : $self.allInstances \longrightarrow forAll(e|e.oclIsTypeOf(ErrorEvent)\ implies\ e.Activity \longrightarrow size() = 1)$

- All error events are intermediate events:

  context *Event*
  inv : $self \longrightarrow select(e|e.type = EventType :: intermediate) \longrightarrow forAll(e.oclIsTypeOf(ErrorEvent))$

- No flow element can be in a direct sequence flow relation with itself:

  context $BPMN_process_fragment$
  inv : $self.fragment_objects.ConnectingObject \longrightarrow select(s|s.oclIsTypeOf(SequenceFlow)) \longrightarrow forAll(s.source \neq s.target)$

- There are no two inclusive gateways in a cycle:

  context *SequenceFlow*
  inv : *let g1 , g2 : InclusiveGateway*
  f : FlowObject
  in
  $(g1.targetFlow = f.sourceFlow)\ and\ (f.targetFlow = g2.sourceFlow)\ implies\ (g2.targetFlow \neq g1.sourceFlow)$

- Any two activities connected by a message flow relation need to belong to different pools:

  context *MessageFlow*
  inv : $self.allInstances \longrightarrow forAll(m|m.from.poolObjects \longrightarrow includes(m.sourceRef)\ and\ m.to.poolObjects \longrightarrow includes(m.targetRef)\ implies\ (m.from \neq m.to))$

- Two flow objects that are in a message flow relation with each other can only be activities or message events:

  context *MessageFlow*
  inv : $self.allInstances \longrightarrow forAll(self.sourceRef.oclIsTypeOf(Activity)\ or\ self.sourceRef.oclIsTypeOf(MessageEvent)\ and\ self.targetRef.oclIsTypeOf(Activity)\ or\ self.targetRef.oclIsTypeOf(MessageEvent))$

- All flow objects with incoming and outgoing flow relations are on a path from the start event to an end event:

  context $BPMN_process_fragment :: Predecesor(x : FlowObject) : FlowObject$
  body : $self.fragment_objects \longrightarrow select(s|s.oclIsTypeOf(SequenceFlow)) \longrightarrow forAll(s|s.target = x\ implies\ result = s.source)$

  context $BPMN_process_fragment :: Succesor(x :FlowObject) : FlowObject$
  body : $self.fragment_objects \longrightarrow select(s|s.oclIsTypeOf(SequenceFlow)) \longrightarrow forAll(s|s.source = x\ implies\ result = s.target)$

  context *FlowObject*
  inv : $self.allInstances \longrightarrow iterate(e : FlowObject;$

$answer : Set(FlowObject) = Set|answer.including(Predecesor(e)) \longrightarrow exists(e : Event|e.type = Event :: Start)$

context *FlowObject*
inv :$self.allInstances \longrightarrow iterate(e : FlowObject;$
$answer : Set(FlowObject) = Set|answer.including(Succesor(e)) \longrightarrow exists(e : Event|e.type = Event :: End)$

- Exception: the only exception to the previous rule refers to the presence of *composition interfaces* - a flow object element can be on a path from a start event that does not finish with an end event, but with a flow object that has a composition tag:

  context *FlowObject*
  inv : $self \longrightarrow iterate(e : FlowObject;$
  $answer : Set(FlowObject) = Set|answer.including(Predecesor(e)) \longrightarrow excludes(e : Event|e.type = Event :: End)\ implies\ answer \longrightarrow exists(a : Activity|a.compositionTag \longrightarrow size() = 1)\ and\ (a.targetFlow \longrightarrow size() = 0)$

- If a flow object has a composition tag, this tag can only be of one type - input or output:

  context *FlowObject*
  inv : $self.composition_tag \longrightarrow size() \neq 0\ implies\ (self.composition_tag.oclIsTypeOf(OutputTag)\ or\ self.compositionTag.oclIsTypeOf(InputTag))$

- An input composition interface only contains flow objects having the input composition tag:

  context *BPMN _process _fragment*
  inv : $self.fragment_interface.Input_interface.elements \longrightarrow select(e : FlowObject|e.compositionTag.IsTypeOf(Output) \longrightarrow size() = 0$

- An output composition interface only contains flow objects having an output composition tag:

  context *BPMN _process _fragment*
  inv : $self.fragmen_interface.Input_interface.elements \longrightarrow select(e : FlowObject|e.compositionTag.IsTypeOf(Input) \longrightarrow size() = 0$

- The sets of composition interfaces of a business process fragment are disjoint:

  context *CompositionInterface*
  inv : $self.InputInterface \longrightarrow asBag() \longrightarrow intersection(self.OutputInterface \longrightarrow asBag) \longrightarrow isEmpty()$

- A flow object cannot be in a sequence flow relation with itself:

  context *BPMN _process _fragment*
  inv : $self.BPMNObject \longrightarrow select(x|x.oclIsTypeOf(SequenceFlow)) \longrightarrow forAll(x.source \neq x.target)$

- A business process fragment is required to have no two sequence flow objects that have the same target and source activities:

  context *BPMN _process _fragment*
  inv : $self.BPMNObject \longrightarrow select(x|x.oclIsTypeOf(SequenceFlow)) \longrightarrow forAll(x_1, x_2 : SequenceFlow|x1.source \neq x2.source\ and\ x1.target \neq x2.target)$

In order to ensure that the OCL constraints presented above are well-written and that they precisely specify the intended well-formedness rules, they need to be verified. For this purpose, we use Dresden OCL, which provides a set of tools to parse and evaluate OCL constraints on various models and meta-models like UML, EMF and Java. Furthermore, Dresden OCL is meta-model independent and can be connected to various meta-models. It offers support for adaptations to MDT UML, EMF Ecore, Java Classes and XML Schema.

5.3 Verification of behavioural correctness of business process fragments

The verification of structural correctness, presented in the previous section, can only allow to check that certain structural properties of the business process fragments are valid. However, we also want to perform checks related to the *dynamic behaviour* of business process fragments. Therefore, we propose the notion of *"behavioural correctness"* which serves to verify the possible behaviours of a business process fragment. Behavioural correctness ensures that a business process fragment does not exhibit any erroneous or unwanted behaviours. The concept is defined based on the original definition of *"process soundness"* proposed by van der Aalst in the context of workflow nets [vdA98]:

Definition: *A workflow-net is sound if and only if:*

- For every state M reachable from state the initial state, there exists a firing sequence leading from state M to the final state;
- The final state is the only state reachable from the initial state with at least one token in the final place;
- There are no dead transitions.

The above definition uses the notion of *firing sequence*, already introduced in Section 2.3.4, and those of *reachability* and *dead transitions* which will be discussed later on in this section.

Based on the notion of soundness, we adapt this concept to the CBPF language and propose a set of properties that should be verified in order to ensure the behavioural correctness of business process fragments. The behavioural properties that should be verified for a business process fragment are separated into two major classes:

- *Generic:* they specify general dynamic properties that any business process fragment should fulfil. Most of them are inspired by the soundness property defined by van der Aalst. Examples of such properties are: reachability of end events or composition interfaces, dead-lock freedom or absence of dead tasks;
- *Fragment specific:* as business process fragments are created to describe a high level functionality or feature, there will exist certain dynamic properties that are specific to each individual fragment and therefore cannot be verified in general. We want to offer the product line engineer the possibility to define and verify such fragment specific properties. Therefore, we propose several general property templates which can be instantiated by the product line engineer for a specific purpose, for verifying a specific property of interest.

Throughout the rest of this section, we describe in detail how the two types of behavioural properties can be verified and how Petri nets can help in this process.

5.3.1 Using HCPN for business process fragment verification

In section 4.4 we provided a formal semantics to the CBPF language using a mapping of concepts to HCPN. The mapping proposed in section 4.4 actually served two purposed:

- Define a formal semantics for the CBPF language in a translational manner;
- Verification of behavioural properties of business process fragments requires advanced analysis techniques. Fortunately, many powerful analysis techniques have been developed for Petri nets [Mur89]. Therefore, the mapping also allows us to take advantage of all the analysis techniques and tools already defined by the Petri net research community.

A more detailed presentation of the main analysis techniques developed for Petri nets which we can take advantage of for verifying the behavioural correctness of business process fragments is available in Section 2.3.4. The abundance of available analysis techniques shows that Petri nets can be seen as a solver independent medium between the design of the business process fragment and its analysis. The type of verifications proposed throughout this section are based on coverability graph (state space) analysis techniques 2.3.4. Some of the questions that can be answered from state space analysis and interesting behavioural properties that can be proven are listed in the following: *boundedness*, *reachability*, *liveness*, *home properties*, *dead markings*, *fairness*. A more detailed description of these properties is presented in Section 2.3.4.

We have seen that HCPN provide powerful analysis techniques that we may use for verification purposes, which we take advantage of due to the mapping provided between CBPF and HCPN. Moreover, another goal of this mapping is to allow access to the already existing analysis and verification tools developed by the HCPN community. These tools can automate the verification process. The tool that we have selected is called *CPN Tools* [JKW07]. It provides an environment for editing and simulating HCPN models, and for verifying their correctness using state space analysis methods. CPN Tools combines powerful functionalities with a flexible user interface, containing improved interaction techniques, as well as different types of graphical feedback which keep the user informed of the status of syntax checks, simulations, etc. The functionalities of the tool that are mostly use in this thesis are the support for two types of analysis for HCP-nets: *simulation* and *state space analysis*. A detailed presentation of CPN Tools, it's functionalities and how it it practically used for verifying HCPN models is given in Chapter 6.

Based on the state space analysis technique available for HCPN and on the behavioural properties which can be verified for HCPNs, listed in Section 2.3.4, we present in the next sections how *general* and *fragment specific* behavioural properties of business process fragment created using the CBPF language can be verified. We also explain how these verifications can be performed with HCPNs.

It should however be mentioned that the verifications we propose in the following are only an initial attempt towards the complete verification of behavioural properties of business

process fragments. We propose a series of general and fragment specific behavioural properties for business process fragments. These properties are then interpreted in terms of Petri net concepts and properties. The actual verification is performed at the Petri net level, in order to take advantage of the rich analysis techniques and tools available at this level. In practice, the CPN Tools package is used for performing the various analysis and verifications on the obtained HCPNs. The feedback that the tool provides is given in terms of Petri net concepts. In most of the cases, we are only interested to know if the property that we are verifying is fulfilled or not. In this case, the result obtained on the HCPN is directly applicable on the original business process fragment and we know if the property is true or false on the process fragment under investigation. However, there might be cases in which CPN Tools will provide additional information when the property that we are checking is not fulfilled, in general a example of execution trace that invalidates the property. This result is meant to help the user in solving the possible errors that might have been detected. However, the changes can only be made at the level of the HCPN. But in the end, we are interested in knowing this information for the business process fragment which is analysed, so that we can modify it accordingly. In order to be able to accomplish this and also interpret some of the more complex feedbacks of the Petri net tool back on the original business process fragment, we need to be able to go back from the HCPN to the original process fragment and point out the possible errors directly at that level. This can be accomplished by defining a *trace model* between the business process fragment and the HCPN that corresponds to it. This trace model can be created during application of the model to model transformation which we proposed between CBPF and HCPN. For the moment, this trace model is part of the future works of this thesis and will be discussed in more detail in Chapter 7. Due to this reason, we acknowledge that an additional step is still required for completing the work on verification of business process fragment behavioural properties.

5.3.2 Verification of general behavioural properties

This category contains a certain set of behavioural/dynamic properties that should hold for all the business process fragments that can be created using the CBPF language. These properties are thus generic and need to be independent of the particularities of the domain or of the SPL to which the business process fragment under analysis belongs to. Therefore, the generic properties that we propose for verification are inspired from the behavioural properties of Petri nets which were described in the previous section. However, these properties need to be adapted to the specific context of business process fragments. In the following, we present a set of generic behavioural properties that we propose for verification and explain how they can be checked using HCPNs:

- **Reachability of end events:** an end event indicates where a business process fragment will finish. Each business process fragments must have at least one end event. We therefore know that by construction, every business process fragment will structurally have at least an end event. However, what this structural property cannot guarantee is that, by executing the business process fragment, the end event will be reached. We therefore want to check if there exists an execution path (execution trace) that begins at the start event of the business process fragment and contains the end event. The property thus guarantees that the business process fragment under analysis *eventually reaches completion*, which may also be referred to as *proper*

completion of the fragment. This means that there exists at least one execution sequence (trace) that reaches the end event, and that when that happens there are no enabled tasks left in the process fragment.

With the help of the proposed mapping between CBPF and HCPN, this property can be easily verified. Based on the mapping templates introduced in Section 4.4.2, and end event is transformed into either a simple place with no outgoing arcs (for the plain end event) or into a special kind of transition (error or message) connected to a a place with no outgoing arcs (for the error and message end events). Therefore, verifying the reachability of the end event of a business process fragment resumes to verifying the reachability of the place with no outgoing arcs in the corresponding HCPN model.

- **Proper completion of a business process fragment:** the previous reachability property ensured that the end events of a business process fragment can always be reached from the start event, which implies that the process instance eventually reached completion. However, we would like to know that when that happens, meaning that the process fragment reaches the end event, there are no enabled tasks still active in the process. This property is called the *proper completion* of a business process fragment.

  The reachability of end events property can be seen as an *option to complete* for the business process fragment. In addition to this, we want to ensure that when the business process ends, it should not have any other tasks still running.

  Using the proposed CBPF to HCPN mapping, verifying this property for a business process fragment resumes to checking that the place with no outgoing arcs (corresponding to the end event) is a *dead marking* and also a *home marking*. The fact that the corresponding HCPN place is a dead marking guarantees the fact that, when this marking is reached, there are no other transitions enabled. Moreover, by asking for that place to also be a *home marking*, we ensure that this place can be reached from any other reachable marking, no matter what happens; this means that the Petri net will always reach this desired state.

- **Reachability of composition interfaces:** composition tags and composition interfaces are two newly introduced important concepts that characterize a business process fragment. As previously discussed, the composition interface defines the exact set of flow objects from a business process fragments where this fragment will be composed with other ones. We know that business process fragments are created with the goal of being later on composed with other fragments. When such a composition occurs, it is the flow object elements tagged with composition interfaces that will be directly concerned by and involved in the actual composition process. Therefore, is is of the utmost importance to ensure that those elements can be reached.

  This means that we need to prove that there is a path (execution trace) that goes from the start event and reaches the tagged flow object. This property is needed for ensuring that, when the business process fragment under analysis is composed with other fragments, the new paths created due to the composition can be accessed from the start event. In case the property is not true for a certain business process fragment, then by composition the resulting fragment will have certain elements that cannot be reached or execution paths that cannot be accessed, which is undesirable. Thus the fulfilment of this property can be seen as a pre-condition for ensuring that the composition of business process fragments take place correctly.

Based on the abstract syntax of the CBPF language, composition tags can only be attached to activities or events. Using the proposed mapping between CBPF and HCPN, we transforms the composition tags from CBPF into similar tags in HCPN. We also know that tagged activities (tasks and sub-processes) are mapped onto the following HCPN constructs: a tagged transition having an input and output place connected to it or a tagged substitution transition having an input and output place connected to it. Based on the mapping templates, tagged events are transformed into places connected by an arc to a tagged transition (either simple or special type - message, error, time). Therefore, verifying this property for a business process fragment resumes to checking that, in the resulting HCPN, the place that follows the tagged transitions (simple, substitution or special) is reachable.

- **Absence of dead tasks:** tasks are atomic activities that define the work performed in a business process fragment. A **dead task** corresponds to a part of the business process fragment that cannot be activated. This means that the presence of dead tasks within a business process fragments implies that there exist some tasks that will not be executed. However, it is desirable that all the tasks of a process fragment be realized, so that the business process fragment can completely execute the behaviour it was meant to perform. Moreover, as a task is meant to be an active element of a business process fragment, a dead task can be interpreted as a passive element which should not appear in the fragment. Thus, the absence of dead tasks means that all the tasks of a business process fragment can be performed.

  Based on the mapping provided between CBPF and HCPN, we know that tasks and sub-processes are transformed into the following HCPN structures: a transition having and input and output place, respectively a substitution transition connected to an input and output place. Therefore, in order to verify that a business process fragment has no dead tasks, we need to check that in the resulting HCPN there are no dead transitions. A transition is considered to be dead if there exist no reachable markings in which that transition is enabled. Dead transitions correspond to parts of the Petri net that can never be activated.

- **Deadlock-free business process fragment:** the concept of deadlock is well known in computer science and describes a situation in which two or more competing actions are each waiting for the other to finish, and thus neither ever does. In the case of business process fragments, a deadlock defines a point in a process fragment that may block the execution of the process. When a process fragment reaches such a deadlock, it cannot continue its execution in any way any further and is thus stuck in that state. This means that, when reaching a deadlock in the process, there are no available outgoing paths that may be taken. The process is stuck in a state from which it is impossible to advance. This is an unwanted situation for any business process fragment that must clearly be avoided for ensuring the correct behaviour of the process fragment. We want to discover deadlocks as early as possible in order to avoid unwanted problems later on. Once a deadlock has been discovered, corrective measures must be taken and the business process fragment under analysis needs to be changed to remove the problem.

  Based on the proposed mapping between CBPF and HCPN, the deadlock-freedom of a business process fragment can be ensured by investigating the absence of dead markings in the corresponding HCPN. A dead marking is one in which no transitions are enable, this no transitions can be fired when that marking is reached. This leads

to the Petri net being stuck in that place, as no transitions can be fired for advancing. We verify the presence or absence of dead markings in a HCPN to check that the net does not run into unwanted deadlocks. However, what we want to prove is that the net does not contain any dead markings, except for the final place of the net. We have seen before that for ensuring the proper completion property of a Petri net, the final place of the net needs to be a dead marking. Therefore, we want to ensure that the final marking is the only dead marking of the HCPN.

- **Compatibility of data type of composition interfaces:** a composition interface explicitly defines the exact place in a business process fragment where the concerned fragment can be composed with other ones. They play a crucial role in the composition of business process fragments. One of the previously presented behavioural properties ensured that composition interfaces are always reachable in a business process fragment. However, we also want to ensure that, when two business process fragments are composed, they are compatible in terms of data. This means that the type of the data output associated to a task tagged with a composition interface belonging to the first fragment is the same as the type of the data input associated to a task tagged with a composition interface belonging to the second fragment. Thus, after the actual composition is performed, the data flow of the resulting fragment is correct.

  This property ensures that two process fragments that are composed are *compatible in terms of data* at the flow objects where the composition will be performed. This also ensures that after composition, the data flow will not be disrupted due to data incompatibility of the elements involved in the composition.

  Based on the mapping between CBPF and HCPN that we propose, this property can be verified by checking boundedness properties on the corresponding HCPNs. The *best lower multi-set bound* of a place specifies for each colour in the colour set of the place the minimal numbers of tokens that is present on this place with the given colour in any reachable marking. This is specified as a multi-set, where the coefficient of each value is the maximal number of tokens with the given value. Best lower multi-set bounds give, therefore, information about how many tokens of each colour that are always present on a given place.

  Therefore, to ensure the compatibility of the composition interfaces of two business process fragments that we want to compose, we must perform the following steps:

  - transform the two business process fragments into corresponding HCPNs;
  - apply the *LowerMultiSet* query function on the place following the concerned tagged transition of the first HCPN;
  - check that the colours corresponding to the data inputs of the tagged flow object belonging to the second process fragment appear in the result obtained in the previous step.

This concludes the presentation of the general behavioural properties which we want to check for all business process fragments. In the following section, we another set of behavioural properties which are specific to the individual business process fragments that are under verification and explain how they can also be checked.

5.3.3 Verification of fragment specific behavioural properties

In the previous section we proposed a set of general dynamic properties that any business process fragment should fulfil. However, certain behavioural properties cannot be verified in general and might differ from one business process fragment to another and depend on the specific SPL context in which that process is used. Business process fragments are created to be the core assets used and consumed by our SPL methodology. They are this created for a specific product line and implement a particular functionality required by some of the products of that SPL. They are thus used in very diverse contexts and implement a variety of functionalities.

We want to offer the product line engineer the possibility to define and verify behavioural properties that are specific to a particular business process fragments and which may only be relevant in a particular context of use. We propose to achieve this by providing the product line engineer with a *set of high-level property templates*, which the product line engineer can then tailor and adapt to his particular needs.

A property template defines in a generic, high-level manner a property that can be verified on a business process fragment. Such templates are defined in a more abstract way and usually take abstract parameters. Then, depending on the specific property that the product line engineer wants to check, he can tailor and adapt the template according to his particular needs. The adaptation of the template to a specific context is done by replacing the abstract parameters by concrete ones. This transforms the template into a specific query function that can be applied, which checks a specific property of interest for the business process fragment under analysis.

As for the generic behavioural properties defined and explained in the previous section, the verification of fragment specific properties is also done using Petri net analysis techniques. Thus, a pre-requisite for all these verification is to apply the transformation from CBPF into HCPN and do all the verifications on the resulting Petri net. Thus, the property templates that we propose throughout this section are defined for HCPN. They are based on standard or non-standard queries that are available in CPN Tools. Checking if they are true or not is also done using CPN Tools.

The proposed property templates are discussed in the following:

- **Can a certain flow object be reached from the start event?**

  In a lot of practical case we are interested to know is, from the start event, a certain flow object can be reached. This means that we want to check if there is at least a path (execution trace) that leads from the start event to a certain flow object. Usually, the flow objects for which we want to perform this verification are activities (tasks or sub-processes). However, the verification can as well be made for events or gateways. In case we perform the check for activities, we are interested to know if that specific activity will be executed or not in the business process fragment.

  In terms of Petri nets, this resumes to proving that a specific node (place or transition) of the resulting HCPN is reachable from the starting place. Therefore, we can use the *Reachable* query function or its chatty version *Reachable'* for creating the property template. Based on these CPN Tools functions, we propose the following property template:

  $Reachable'(id - node_{start}, Var : id - node_{interested})$

This is just a simple application of the standard *Reachable'* query function for which the first parameter used is a constant defining the id of the start place of the net (usually 1), and the second parameter is a variable denoting the id of the node for which we want to check the reachability. The template can easily be used for performing a particular query by simply replacing the variable parameter with the specific value of the place for which we perform the check.

A simple exemplification of the use of this property template is given in the following. Based on the mapping templates we proposed, we know that tasks are mapped onto the following HCPN structure: place connected by an arc to a transition with the name of the task connected then by another outgoing arc to a place. In order to check that a task can be reached form the start event, we simply have to apply the following query on the resulting HCPN: $Reachable'(1, id_{task})$, where id_{task} denotes the id of the place from the net that follows the transition with the same name as the task. The function will either return false in case the task is not reachable, or it will return true followed by an example possible path.

- **Is a certain task always executed?**

  In a business process fragment, tasks are the atomic units of execution. However, certain tasks may be of more importance for defining the behaviour of the fragment than others. Also, certain tasks may be critical for the execution of the fragment and thus are more important than the other tasks present in the fragment. Therefore, we want to offer the product line engineer the possibility to check that such crucial tasks which are of special interest are always executed in every possible execution of the business process fragment under investigation. This means that such a task is part of all the execution traces of the business process fragment.

  In terms of Petri net concepts, this resumes to checking that a certain place of the resulting HCPN is or not a home marking. A home marking is characterized by the fact that it is reachable from every other marking, no matter what happens. This means that the HCPN will always reach this desired place. We can thus use the *HomeMarking* or *ListHomeMarkings* query functions provided by CPN Tools. Based on these CPN Tools functions, we propose the following property template:

  $HomeMarking(Var : id - node_{interested})$

  The template is just a simple application of the standard *HomeMArking* query function for which the only parameter is defined as a variable denoting the id of the place which we want to prove is a home marking.

  A simple exemplification of the use of this property template is given in the following. Based on the mapping templates propose, we know that a task is transformed into the following HCPN structure: a place connected by an arc to a transition that has the same name as the CBPF task connected by an outgoing arc to another place. Thus, in order to check that a certain task can always be executed for the current business process fragment, we simply need to apply the proposed property template in the following manner on the resulting HCPN: $HomeMarking(id_{task})$, where id_{task} denotes the id of the place from the HCPN that follows the transition with the same name as the task. The function will either return false in case the task is not always executed, or it will return true in case the property is fulfilled.

- **Will a certain task have data objects of a particular type?**

We know that data inputs and data outputs are assigned through data associations to tasks in order to model the data representation and data flow for a business process fragment. The product line engineer might be interested to know if, during the possible executions of the business process fragment, a specific task will contain or not data objects of a specific type. We know that the data input of a task defines the particular types of data required for the execution of that task. However, we do not know if, during the execution of the business process fragment, those data types will be available at that task in order to activate it. This is what this property allows us to determine.

In terms of Petri net concepts, this resumes to checking the colours of all the tokens that may arrive in a certain place and see if the desired colour or colours are among them. Determining the colours of the tokens that may arrive in a specific place throughout all the possible executions of the net can be done using the boundedness Petri net property. We propose the following property template:

$LowerMultiSet(Var : id - node_{interested})$

The template is just a simple application of the standard *LowerMultiSet* query function for which the parameter is defined as a variable denoting the id of the place for which we want to determine the possible colour sets.

A simple exemplification of the use of this property template is given in the following. Based on the mapping templates propose, we know that a task is transformed into the following HCPN structure: a place connected by an arc to a transition that has the same name as the CBPF task connected by an outgoing arc to another place. Also, the data objects associate to a task are transformed into colours and colour sets. Thus, in order to check that a certain task will have data objects of a certain type during the execution of the business process fragment, we simply need to apply the proposed property template in the following manner on the resulting HCPN: $LowerMultiSet(id_{task})$, where id_{task} denotes the id of the place from the HCPN that precedes the transition with the same name as the task. The function will return a set of tokens and their colours. The product line engineer needs then to check if the data type that we are checking for, which corresponds to a specific colour, is included in the previously obtained set. In this is the case, we can say that the task under investigation will have that particular type of data as input during the business process fragment execution.

- **If a particular activity x is executed, then will activity y also be executed?**

  In certain cases and for certain business process fragments, there may exist activities that are logically connected. This means that when one of them is executed, then the other one needs also to be executed. Thus there is a certain dependency between them. This property allows the product line engineer to check, for two activities of the business process fragment, knowing that one of them is executed, if the other one is also executed. We therefore want to see that, if there is a path (execution trace) that goes from the start event until activity x, then if that path also contains activity y.

  In terms of Petri net, this resumes to checking if a certain node belongs or not to an execution trace that contains another node. We propose the following property template:

  $NodesInPath(id - node_{start}, Var : id - node_2)$

The template is an application of the standard *NodesInPath* query function for which the first parameter is a constant denoting the id of the start place of the resulting net (usually 1), and the second parameter is a variable denoting the id of one of the places that needs to be in the path.

A simple exemplification of the use of this property template is given in the following. Based on the mapping templates propose, we know that a task is transformed into the following HCPN structure: a place connected by an arc to a transition that has the same name as the CBPF task connected by an outgoing arc to another place. We have two cases for our property:

- *Task x executed before task y:* this means that we want to check that there exists a path between the start event and task y, and that task x is part of this path. That will guarantee that when task y is executed, then task x has already been executed before it. In order to verify this, we need first to apply the property template in the following way: $NodesInPath(1, id - node_y)$, where $id - node_y$ is the id of the place following the transition named "y". The function will return a set of nodes that represent a path between the start event and task "y". Once this is done, we need to check that node "x" is part of the result previously obtained.
- *Task x executed after task y:* his means that we want to check that there exists a path between the start event and task x, and that task y is part of this path. That will guarantee that when task x is executed, then task y has already been executed before it. In order to verify this, we need first to apply the property template in the following way: $NodesInPath(1, id - node_x)$, where $id - node_x$ is the id of the place following the transition named "x". The function will return a set of nodes that represent a path between the start event and task "x". Once this is done, we need to check that node "y" is part of the result previously obtained.

- **If activity x is executed then activity y will never be executed**

  In some cases, some activities of a business process fragment might be mutually exclusive. This means that when one of them is executed, the other one will implicitly never be executed. We want to offer the product line engineer the possibility to check this type of situations in a business process fragment and verify if two activities which are of interest for a business process fragment are mutually exclusive. This type of verification is important as we may want to check that it is never true for a pair of activities and there is thus always an execution path connecting them, or on the contrary to verify that the property is always true for a pair of activities which need to be mutually exclusive.

  In terms of Petri net concepts, this resumes to verifying that there exists or not a path (execution trace) connecting the two activities. In other words, proving that one activity is reachable from the other. We propose the following property template:

  $Reachable'(Var : id - node_2, Var : id - node_2)$

  The template is an application of the standard *Reachable'* query function for which both parameters are variables denoting the id of the tasks that we want to check are connected by a path.

  A simple exemplification of the use of this property template is given in the following. Based on the mapping templates propose, we know that a task is transformed into

the following HCPN structure: a place connected by an arc to a transition that has the same name as the CBPF task connected by an outgoing arc to another place. Thus, in order to verify that if activity "x" is executed then activity "y" will never be executed, we simply need to apply the property template in the following manner: $Reachable'(id - node_x, id - node_x)$, where $id - node_x$ and $id - node_x$ correspond to the ids of the places situated right after the transitions named "x" and "y" in the resulting HCPN. In order for the property to be fulfilled, the functions needs to return false. Otherwise, there exists a path between the two activities and the property is invalidated.

- **User defined property templates**

  Throughout this section we presented a set of property templates which the product line engineer can use and tailor for verifying different business process fragment specific behavioural properties. However, the templates we propose will surely not cover all the possible verifications that a product line engineer might want to perform on a business process fragment. Thus, we want to offer the possibility to the product line engineer to write his own queries, adapted for verifying a particular fragment specific behavioural property which cannot be checked with any of the templates proposed.

```
1  ;;*************************************************************
2  ;; Generic node search
3  ;;*************************************************************
4
5  SearchNodes (Area, Pred, Limit, Eval, Start, Comb) where:
6
7  area search area Node list
8  pred predicate function Node -> bool
9  limit search limit int
10 eval evaluation function Node -> 'a
11 start start value 'b
12 comb combine function 'a * 'b -> 'b
```

Fig. 5.1: Generic query function for node search and processing

CPN Tools offers a generic, highly parametrizable function that allows to search and perform various processing operations on the nodes of the occurrence graph. The function is described in Figure 5.1. The generic query function *SearchNodes* traverses the nodes of the occurrence graph. At each node some specified calculation is performed and the results of these calculations are combined, in some specified way, to form the final result. The function takes six different arguments and by varying them it is possible to specify a lot of different queries:

- *Search Area:* specifies the part of the occurrence graph which should be searched. It is often all nodes, but it may also be any other subset of nodes;
- *Predicate function:* this argument specifies a function. It maps each node into a boolean value. Those nodes which evaluate to false are ignored, while the others take part in the further analysis;
- *Search limit:* specifies an integer which tells us how many times the predicate function may evaluate to true before we terminate the search. The search limit may be infinite. This means we always search through the entire search area;

- *Evaluation function:* specifies a function that maps each node into a value, of some type A. The evaluation function is only used at those nodes (of the search area) for which the predicate function evaluates to true;
- *Start value:* this argument specifies a constant, of some type B;
- *Combination function:* this argument specifies a function that maps from $A \times B$ into B, and it describes how each individual result (obtained by the evaluation function) is combined with the prior results.

By convention, the following values are used for some of the parameters:

- *val* *EntireGraph* to denote the set of all nodes in the occurrence graph;
- *val* *NoLimit* to specify an infinite limit for the search limit;

A pseudo-code like description of how the *SearchNodes* function works is presented in Figure 5.2.

```
1
2  SearchNodes (Area, Pred, Limit, Eval, Start, Comb)
3  begin
4     Result := Start; Found := 0
5      for all n in Area do
6       if Pred(n) then
7        begin
8         Result := Comb(Eval(n), Result)
9         Found := Found + 1
10        if Found = Limit then stop for-loop
11       end
12     end
13 end.
```

Fig. 5.2: Pseudo-code description of *SearchNodes* function

The *SearchNodes* function is a bit complicated to understand and use. However, it is also extremely general and powerful. It can actually be used to implement most of the standard query functions that were presented in this section. The product line engineer can thus use this function for preforming different business process fragment specific verifications by adapting this generic function and applying it to the HCPN corresponding to the business process fragment under analysis.

This concludes the presentation of the different general or fragment specific behavioural properties that we propose for verifying the behavioural correctness of a business process fragment.

6. EXEMPLIFICATION OF THE PROPOSED METHODOLOGY AND TOOL SUPPORT

Abstract

Throughout this chapter we exemplify the SPL methodology proposed in Chapter 3 by applying it to a case study from the crisis management system domain. This case study also servers to facilitate the understanding of the concepts and the functioning of the CBPF language proposed in Chapter 4, but also to exemplify the verification techniques of business process fragments proposed in Chapter 5. We start by introducing the bCMS car crash crisis management system case study in Section 6.1. Then, throughout Section 6.2, we apply the proposed SPL methodology on the bCMS case study. We follow the methodology as it was introduced in Chapter 3 and, for each of its steps, explain and exemplify how it applies on the bCMS case study. In the second part of the chapter, in Section 6.3, we present the SPLIT tool suite, which is the tool support that we propose for our methodology. Good tool support is one of the key elements for the fast adoption of any new methodology and language. We start by describing the general requirements that such a tool should fulfil. We then present the general architecture of the proposed tool and discuss in more details the different tool modules and the functionalities each of them provides.

6.1 Introducing the bCMS case study

Throughout this section we introduce the *"bCMS car crash crisis management system"*, which serves the following purposes:

- explain and exemplify the SPL methodology presented in Chapter 3;
- facilitating the understanding of the concepts and the functioning of the CBPF language proposed in Chapter 4;
- exemplify the verification of business process fragments proposed in Chapter 5.

The initial bCMS case study [CCG+11] was defined at the 2011 AOM Bellairs Workshop on Developing End-to-End AOSD Artifacts [1]. The case study was then improved during the Comparing Modeling Approaches (CMA) workshop [2] at MODELS 2011, becoming a focused case study that defines a simple Crisis Management System.

[1] http://www.cs.mcgill.ca/ joerg/SEL/AOM-Bellairs-2011.html
[2] http://cserg0.site.uottawa.ca/cma2011/index.htm

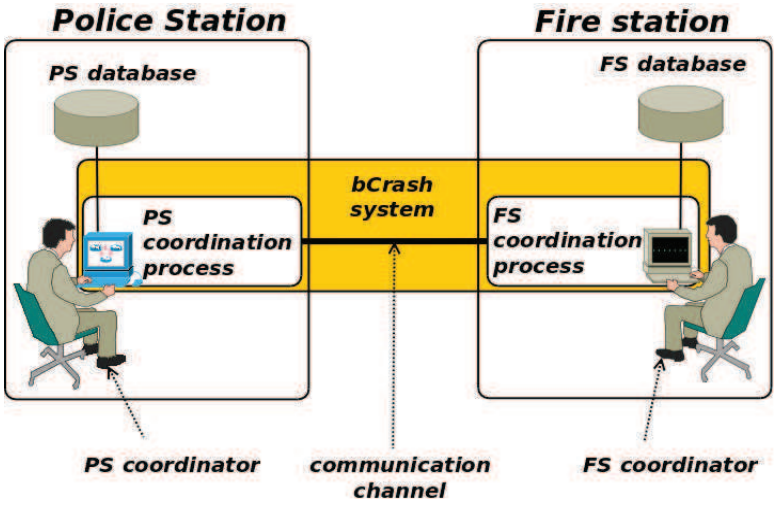


Fig. 6.1: Overall view of the environment and the desired system

The purpose of the bCMS document [CCG+11] is to define the requirements of a Software Product Line (SPL) called *bCMS-SPL* and aimed at managing car crash crisis. Basic features along with desired variations are proposed such that it results in a small SPL definition. The primary focus of the proposed variations is to allow for static and dynamic variations (i.e., dynamic change between variants at runtime). The software product line is described in the following manner: the specification of a "reference variant" of the SPL referred to as bCMS is first provided; in a specific section, we then include all the information concerning possible variations that could be applied to bCMS. In this way, all the variation points and their possible implementations are introduced. A detailed description of the bCMS case study can be found in [CCG+11].

The bCMS requirements definition document is structured in the following manner. It introduces first the scope and stakeholders of the system. Then, functional and non-functional requirements are specified. A discussion about hardware and standards and a definition of the allowed variation points follows. The document concludes with a data dictionary and a glossary.

The bCMS system is a distributed crash management system responsible for coordinating the communication between a fire station coordinator (FSC) and a police station coordinator (PSC) to handle a crisis in a timely manner (as seen in Figure 6.1). Internal communication among the police personnel (including the PSC) is outside the scope of the desired system. The same assumption applies to the fire personnel (including the FSC). Information regarding the crisis as it pertains to the tasks of the coordinators will be updated and maintained during and after the crisis.

There are two collaborative sub-systems. Thus, the global coordination is the result of the parallel composition of the (software) coordination processes controlled by the two distributed coordinators (i.e., PSC and FSC). There is no central database; fire and police stations maintain separate databases and may only access information from the other database through the bCMS system. Each coordination process is hence in charge of adding and updating information in its respective database.

bCMS starts operating at the point when a given crisis has been detected and declared both at the fire station and the police station, independently. The coordinators (i.e., PSC and FSC) have already defined the parameters necessary to start handling the crisis. The initial emergency call of a witness and any subsequent notifications of the crisis from additional witnesses through either the police station and/or fire station call centers are outside the scope of the desired system.

There is a specific section in the bCMS document that discusses the stakeholders of the system and states the objective of each one:

- *Fire Station Coordinator (FSC)*: maintains control over a crisis situation by communicating with the police station coordinator (PSC) as well as firemen;
- *Fireman:* acts on orders received from the FSC and reports crisis-related information back to the FSC. Furthermore, he communicates with other firemen, victims, and witnesses at the crisis location;
- *Police Station Coordinator (PSC)*: maintains control over a crisis situation by communicating with the fire station coordinator (FSC) as well as policemen;
- *Police officer*: acts on orders received from the PSC and reports crisis-related information back to the PSC. Furthermore, he communicates with other policemen, victims, and witnesses at the crisis location;
- *Victim*: has been adversely affected by the crisis and may communicate with policemen and firemen;
- *Witness*: has observed the crisis and communicates with policemen and firemen;
- *Government agencies*: provide funding for the system and expect improvements of the communities' living standard from the deployment of the system.

The functional requirements of the system are detailed in a separate section in terms of use case of the bCMS system. Also, another section briefly discusses three non-functional requirements.

The purpose of the *"Variations"* section of the requirements document is to define the requirements for the bCMS-SPL. The approach chosen for describing the SPL is to define and detail the possible variations points that could be applied to the "reference variant" bCMS, which is described in the other sections of the bCMS requirements document. Desired variations are proposed that result in a small SPL definition. The primary focus is to allow for static and dynamic variations. The variations proposed cover both functional and non-functional requirements variations. Furthermore, for each variation two priorities are defined. A priority may either be "must have" (i.e., the variation must be part of the model) or "may have". The proposed variation points are the following:

- *Police and Fire Stations Multiplicity*;
- *Vehicles Management*;
- *Vehicles Management Communication Protocol*;
- *Crisis Multiplicity*;

- *Confidentiality of Data Communication*;
- *Authentication of System's Users*;
- *Communication Layer*;

For a better understanding of the bCMS system, we present in the following a structural view that describes the key elements of the system and their relationships. The class diagram notation is used to construct a domain model of the system that describes the elements of the system and the portion of the environment with which those elements interact. For example, while the *PS coordinator* and F*S coordinator* are the key elements of the bCMS system, the *PS coordinator* needs to interact with individual *police units* that are also modelled, but their detailed functionality is beyond the scope of the system. Therefore, the individual police unit is included in the domain model to provide context for the system elements serving as an entity to receive and send information to the *PS coordinator*. The key software elements of the domain model have been described in a data dictionary that is available in [CCGI11]. The domain model is graphically presented in Figure 6.2.

6.2 Applying the proposed SPL methodology on the bCMS case study

In Chapter 3 we proposed a new software product line engineering methodology that focuses on the derivation of product behaviour. The methodology covers the entire SPLE process, from variability modelling and core assets definition during the domain engineering step all the way to the actual product derivation during application engineering. By applying this methodology, we can produce behavioural product models that belong to the analysis and early design levels of the software development life-cycle. The proposed methodology covers only the derivation of behavioural product models, and does not address the structural product representation.

Throughout this section we apply the proposed SPL methodology on the bCMS case study. This will both facilitate the understanding of the different phases of the methodology (how they are actually applied) and point out the string points and limitations of the proposed methodology. We follow the methodology as it was introduced in Chapter 3 and, for each of its steps, explain and exemplify how it applies on the bCMS case study.

6.2.1 Construction of the feature diagram

The first step of the methodology focuses on defining the system properties relevant to the stakeholders and also on capturing the common aspects and those that discriminate among systems in the product family. To achieve this goal, we use *feature models*, a popular SPL variability modelling technique. We will therefore create a feature diagram of the bCrash SPL system, based on the requirements document provided. The process is quite difficult due to the fact that the information that needs to be extracted, the domain knowledge, resides in natural language requirements documents and user specifications.

The feature diagram of the bCMS SPL is created based on the requirements document. It is created in a two steps process:

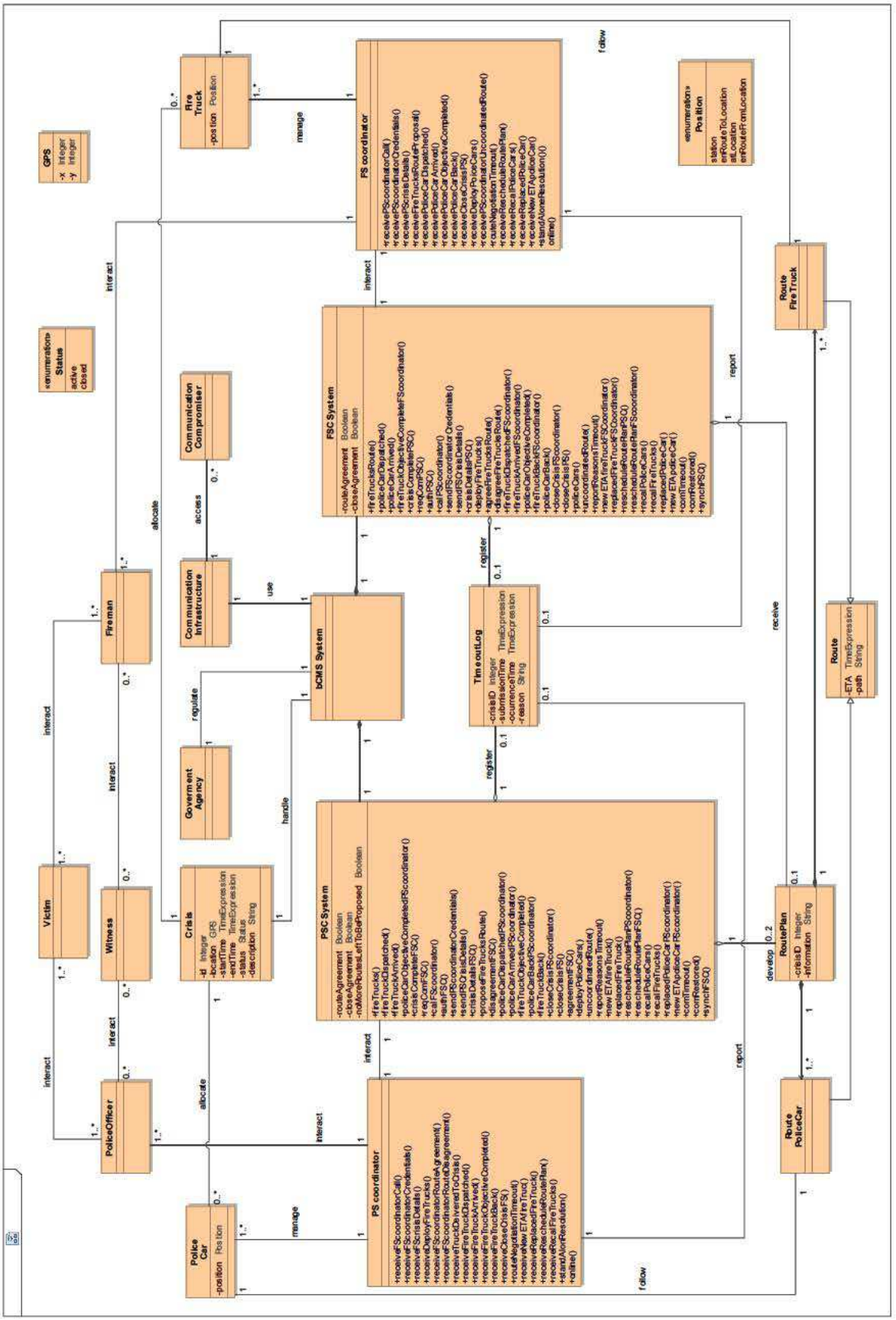


Fig. 6.2: Domain model of the bCMS system

- analysis the possible variation points listed in Section 7 of the bCMS requirements document [CCG+11];
- analysis of the "reference variant", in order to identify the features corresponding to the basic actions defined in the given scenario. For this purpose we analyse both the functional and non-functional requirements of the "reference variant", corresponding to Sections 4 and 5 of the requirements document, respectively.

Analysis of the variation points:

In the requirements document [CCG+11], variations are divided into functional (section 7.1 to 7.4) and non-functional (section 7.5 to 7.7) depending the type of requirement they target. We can start constructing the feature diagram from its *root feature* named *bCMS*. Moreover, the feature diagram of the bCMS system must have *two mandatory features* called *Functional* and *Non-functional*, which descend of the root feature. An AND feature group decomposition is used to relate these two features with the root feature.

The priorities "must have" and "may have" defined in the requirements document are translated as mandatory and optional features into the feature diagram, respectively. The concrete variants to be implemented by each variation point correspond to those items listed within the "Variations" part of each variation point. Similarly, the information placed in the "Constraints" part is used to determine the type of relationship between the different concrete variants of a variation point. This information leads towards the creation of the following features:

- *Police and Fire Stations Multiplicity:* mandatory feature whose parent is feature *Functional*. It is decomposed into two mutual exclusive (XOR) features named *One PS FS* and *Many PS FS*.
- *Vehicles Management:* optional feature whose parent is also feature *Functional*. It is decomposed into two mutual exclusive variants. In the constraints part of Section 7.2, it is indicated that the variant *No send receive* excludes all the other variants (*PSC send receive*, *FSC send receive*, *PSC FSC send* and *PSC receive*). Thus, for easing the modelling, variants 2â5 have been regrouped in a unique variant called *Other*, which is mutually exclusive with variant *No send receive*. In this way, the first constraint is fulfilled. Variants 2..5 defined in Section 7.2 become children of feature *Other* and are all optional, related by an AND decomposition relation. The second and third constraints are represented as "require" feature dependencies between the different variants.
- *Vehicles Management Communication Protocol:* optional feature whose parent is feature *Functional*. It is decomposed into two optional sub-features named *SOAP* and *SSL*. These features are related to each other by an AND decomposition relation. The phrase "In case the system offers the functionality of communication between PSC/FSC and their respective vehicles" is interpreted as a "require" feature dependency between these features and the feature *Other* from the feature *Vehicles Management*.
- *Crisis Multiplicity:* mandatory feature whose parent is feature *Functional*. It is decomposed into two mandatory features named *Single* and *Multiple*. These two features are related by a mutual exclusion (XOR) feature decomposition relation.

- *Communication Layer:* optional feature whose parent is feature *Non-functional*. It is decomposed in two mutually exclusive features named *Proprietary* and *Other*. Feature *Other* is introduced in order to fulfil the first constraint given in Section 7.7. Concrete variants *HTTP* and *SOAP* are children of feature *Other* and are connected to each other by an AND feature decomposition relation.

- *Authentication of System's Users:* optional feature whose parent is feature *Non-functional*. It is decomposed into five optional features which correspond (and are named after) to the variations defined in section 7.6. There is a further decomposition of feature *Challenge response* into three mutual exclusive sub-features called *Symmetric encryption*, *Mutual authorization*, and *Kerberos*.

- *Data Communication Confidentiality:* mandatory feature whose parent is feature *Non-functional*. It is decomposed into two mutually exclusive sub-features named *Encrypted* and *Not encrypted*.

Analysis of the reference variant:

As the requirements document specifies, Sections 4 and 5 describe the characteristics of a "reference variant of the SPL". Therefore, the analysis of these sections will produce a set of mandatory features that should be captured in the feature diagram.

Based on the main scenario defined in Section 4, mandatory features corresponding to the basic actions defined in this scenario can be extracted. These features become children of the *Functional* feature previously defined. As a result of this analysis, the following mandatory features are extracted:

- *communication establishment*;
- *coordinator identification*;
- *crisis details exchange*;
- *coordinate route plan creation*;
- *vehicle dispatch coordination*;
- *vehicle target arrival coordination*;
- *objective completion coordination*;
- *vehicle return coordination*;
- *close crisis*;

The same kind of analysis is performed now on Section 5 from the requirements document, and results in a set of features that become mandatory children of feature *Non-functional*. These mandatory features are related to each other by an AND feature decomposition, are the following:

- *integrity*;
- *availability*;

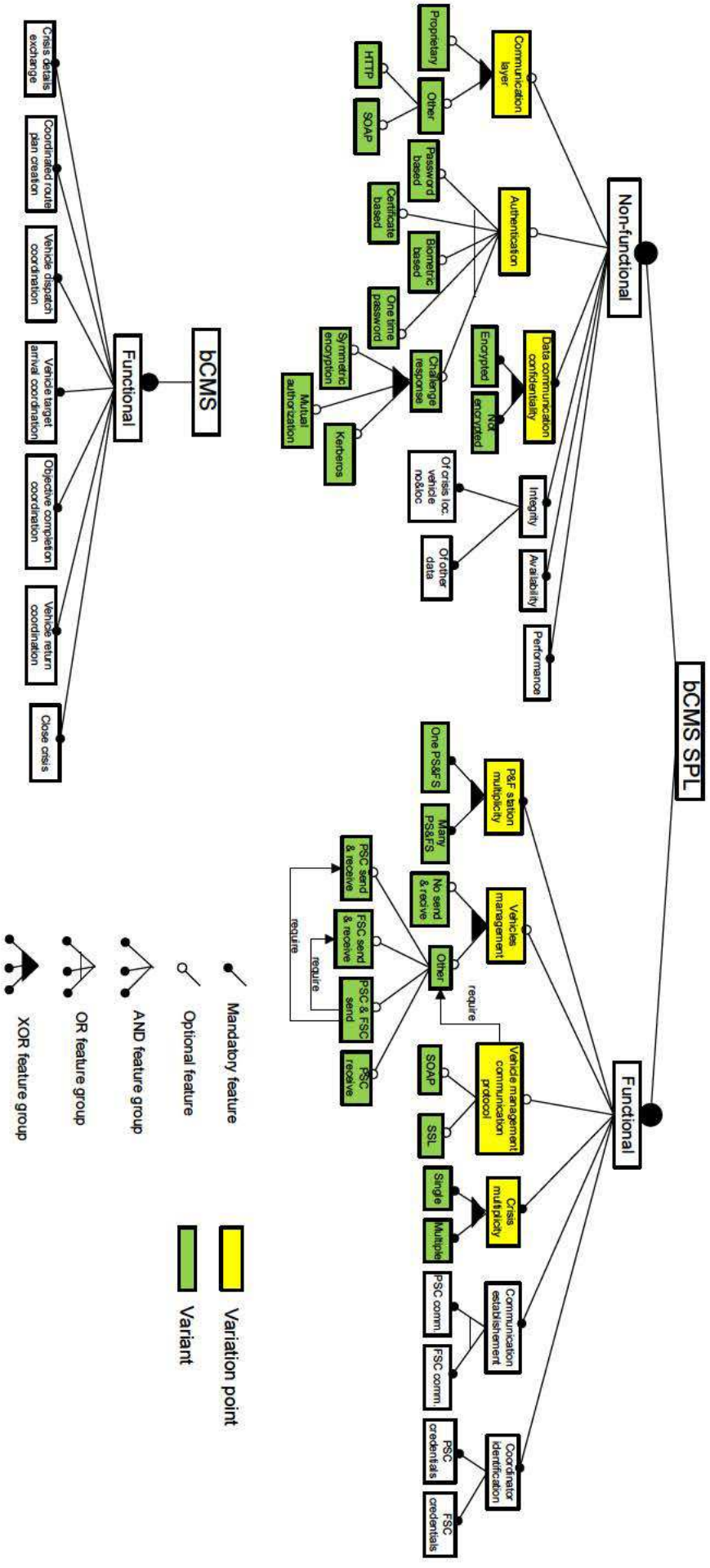


Fig. 6.3: Complete feature diagram of the bCMS system

- *performance.*

The end result of this entire process is the feature diagram of the bCMS SPL system, presented in Figure 6.3. To make its understanding easier, the features coloured in yellow represent variation points, while the variants of a variation point are coloured in green.

6.2.2 Creation of business process fragments

The second step of the methodology consists in creating *business process fragments*, which are the core assets used by the methodology. We presented in Section 3.3 two main ways in which business process fragments can be created:

- *Construct a new process fragment:* a business process fragment can be seen as a detailed specification of a high level abstract task or functionality. Therefore, new business process fragments can be created from scratch as concrete implementations of features from the feature diagram. The CBPF language presented in Chapter 4 is used for creating and modelling business process fragments from scratch.
- *Reuse existing process fragments:* another possible way to obtain business process fragments is by *reusing existing ones*. For this, a business process repository/library is required. There are two distinct ways in which a process fragment from a business process repository can be reused in our methodology:
  - *Reuse process fragment as-is:* corresponds to an "of the shelf" reuse of process fragments. The product line engineer will select, based on the requirements and the description of the functionality that needs to be implemented, a business process fragments form the process repository that best fits the requirements. The selected process fragments is directly used as-is.
  - *Adapt existing process fragment:* reuse an existing process fragments by adapting and tailoring it to the specific requirements of the functionality we need to implement.

For the bCMS case study, we will construct the necessary business process fragments from scratch, following the first construction method explained above. We create new business process fragments from scratch, using the CBPF language, as concrete implementations of features from the bCMS feature diagram obtained previously. This construction process is highly based on the information available in the bCMS requirements document [CCG+11], form where the functional and non-functional requirements for the fragment will be extracted. The knowledge and expertise of a domain expert was required and highly improved the quality of the resulting business process fragments. To create the business process fragments, we need to identify the key functionalities that the process fragment has to implement and to express this information in a concise, flexible and reusable manner.

For a complete description, we need to create a business process fragment for each feature of the bCMS feature diagram presented in Figure 6.3. This is the starting point for constructing the business process fragments. The name of the features provide the abstract, high-level functionality that the corresponding business process fragments need to actually implement. However, this will not suffice for creating accurate and well-designed business process fragments.

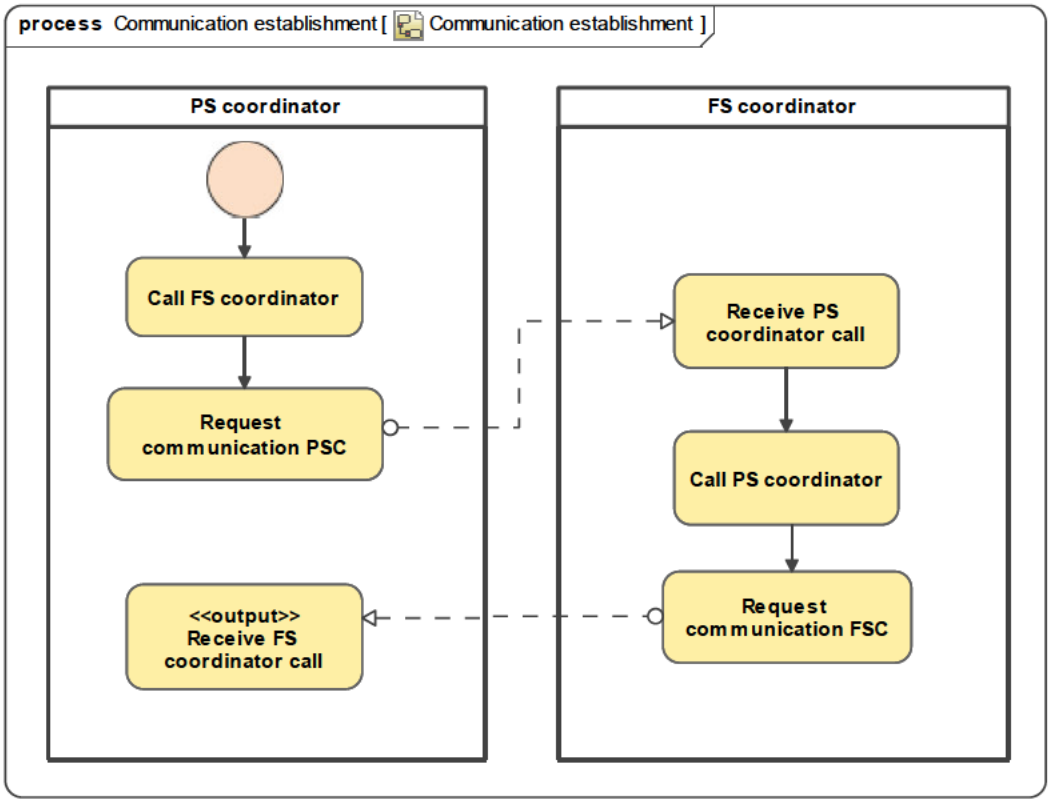


Fig. 6.4: Communication establishment business process fragment

For a complete understanding of the exact functionality that each business process fragment needs to provide, we need to closely study the requirements document. Section 4 of the document describes the *functional requirements* of the bCMS SPL. They are detailed in the form of use-cases (main scenario plus alternative and exceptional scenarios). Based on these descriptions, we can infer the behaviour of a large part of the bCMS features, those that are children of the *Functional* feature: *communication establishment*, *coordinator identification*, *crisis details exchange*, *coordinate route plan creation*, *vehicle dispatch coordination*, *vehicle target arrival coordination*, *objective completion coordination*, *vehicle return coordination* and *close crisis*. The business process fragments corresponding to some of these features are presented and discussed in the following. However, due to their large number and the fact that they are mostly created in the same manner, we only exemplify some of them here. The complete set of business process fragments can be found in Annex 2.

The first business process fragment that we create is called *communication establishment* and corresponds to the feature with the same name. It is graphically depicted in Figure 6.4. We know from the requirements document that the goal of this process fragment is to model how the *police station coordinator* and the *fire station coordinator* establish contact between themselves and start the communication. We can notice from the figure that there are two vertical pools having the names of the main actors involved in this process. The role of the pools is to model the two roles: *police station coordinator* and *fire station coordinator*. The behaviour described is simple: the PS coordinator calls the FS coordinator and tries to establish communication; once the FS coordinator receives the call, he will will also try to contact the PS coordinator in response. What should be noticed is that the business process fragment ends with a task (*receive FS coordinator call*) tagged with an *output composition tag*. The fact that the fragment ends with a tagged task and not with an end event is a specific characteristic of business process fragments. This has two goals: show that the business process fragment models partial information and that the fragment will be completed, by composition, with the necessary information. The composition tag also defines that, when the fragment is composed with other ones, the actual composition will be performed at this exact place and that the fragment will be extended below this place.

A more complex business process fragment that we present here is called *creation of coordinated route plan*, corresponding to the feature with the same name from the bCMS feature diagram. The behaviour described by this fragment is a negotiation between the PS coordinator and the FS coordinator for establising a common plan for deploying their respective police cars and fire trucks. The PSC and the FSC announce each-other that they want to deploy their respective vehicles for intervention at the crisis location. It is the PSC that proposes a common route plan to the FSC. In case the FSC agrees with the proposed plan, he sends his acknowledgement back to the PSC and the fragment ends. In case he does not agree, he proposes an alternative route and sends this information to the PSC. It is now the turn of the PSC to analyse the newly proposed route. In case of agreement, he confirms this to the FSC. In case he does not agree with the new route, then the negotiation reaches a time-out state and the business process fragment ends with an error. This business process fragment is graphically depicted in Figure 6.5.

Finally, the last business process fragment which we present here, created based on the *functional requirements* of the bCMS SPL, is called *close crisis*. It corresponds to the feature with the same name from the bCMS feature diagram. It describes how the PS coordinator and FS coordinator communicate to each other that the crisis has been solved and thus they agree that it should be ended. They coordinate and communicate to each

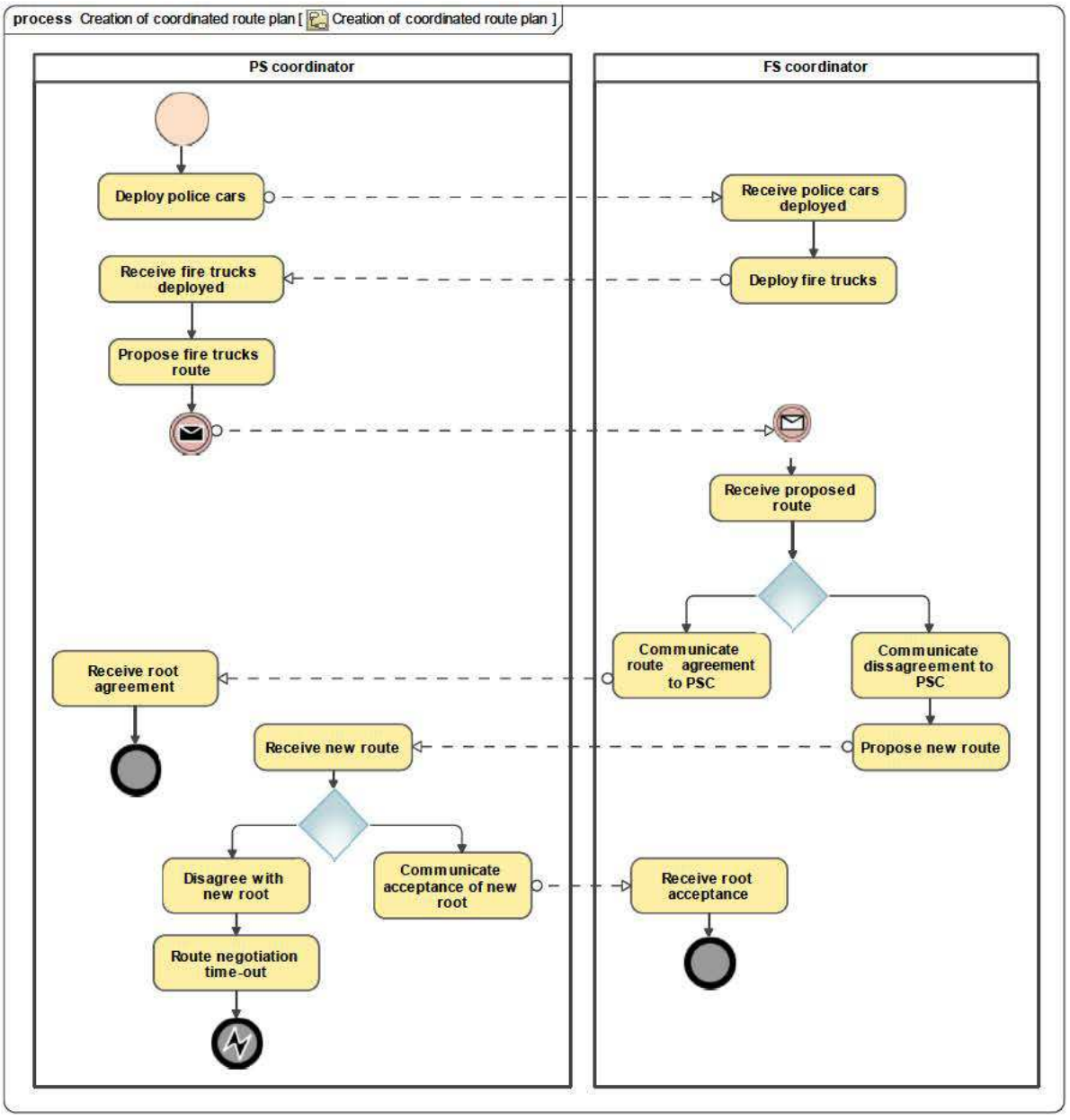


Fig. 6.5: Creation of coordinated route plan business process fragment

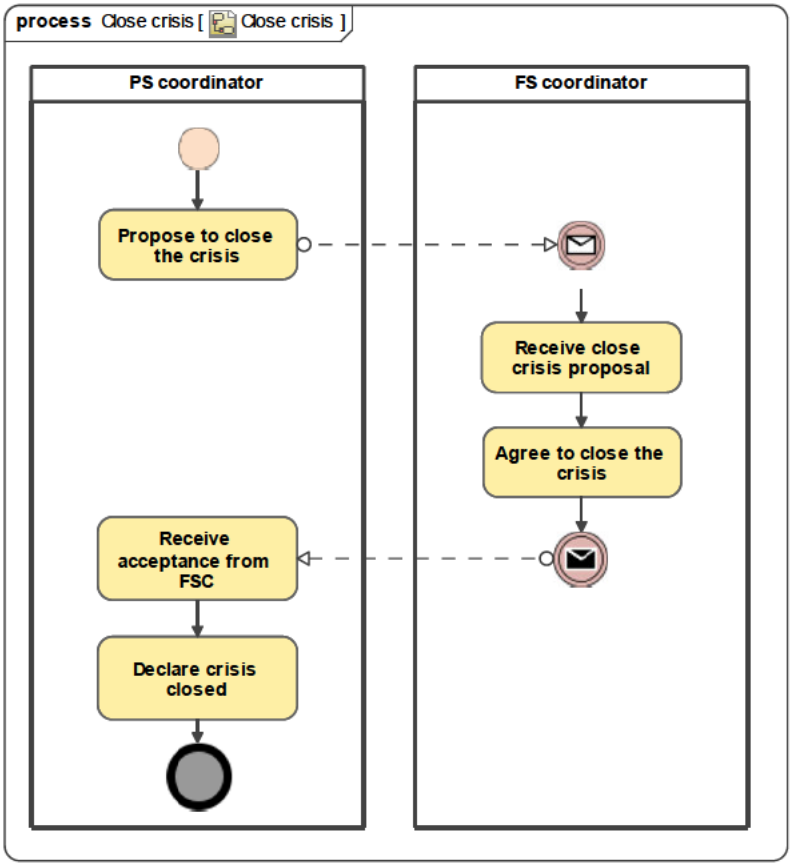


Fig. 6.6: Closing the crisis business process fragment

other this decision. The business process fragment is graphically described in Figure 6.6.

In the feature diagram of the bCMS system from figure 6.3, there are other features besides the ones presented above that are children of the feature *Functional*. Those features have been created based on section 7 of the requirement document. Therefore, we also need to create new business process fragments that implement these features. These fragments are created based on the *"Variations"* section from the bCMS requirements document. Some of them are presented in the following. The complete list of business process fragments is available in Annex 2.

A first business process fragment that we present is called *"PSC send and receive"* and corresponds to the feature with the same name, which is a variant of the *Vehicle management* variation point. It is a simple fragment that describes how the PS coordinator, by using a dispatch service, broadcasts the dispatch order to the concerned police cars. The fragment is graphically depicted in Figure 6.7.

Another business process fragment that we present here is called *"Multiple Crisis"* and corresponds to the feature with the same name, which is a variant of the *Crisis multiplicity* variation points. The business process describes how the PSC and the FSC deal with the fact that multiple crisis may exist. Thus the PSC will selects a certain crisis to be addressed and proposes to the FSC to intervene on this crisis. If the FSC agrees, it sends its acknowledgement to the PSC and the fragment ends. In case the FSC does not agree

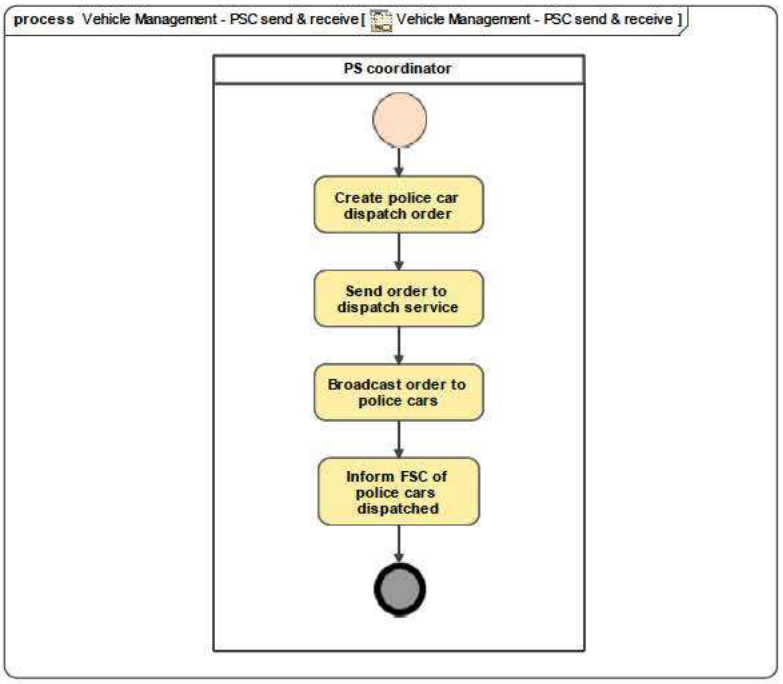


Fig. 6.7: PSC send and receive business process fragment

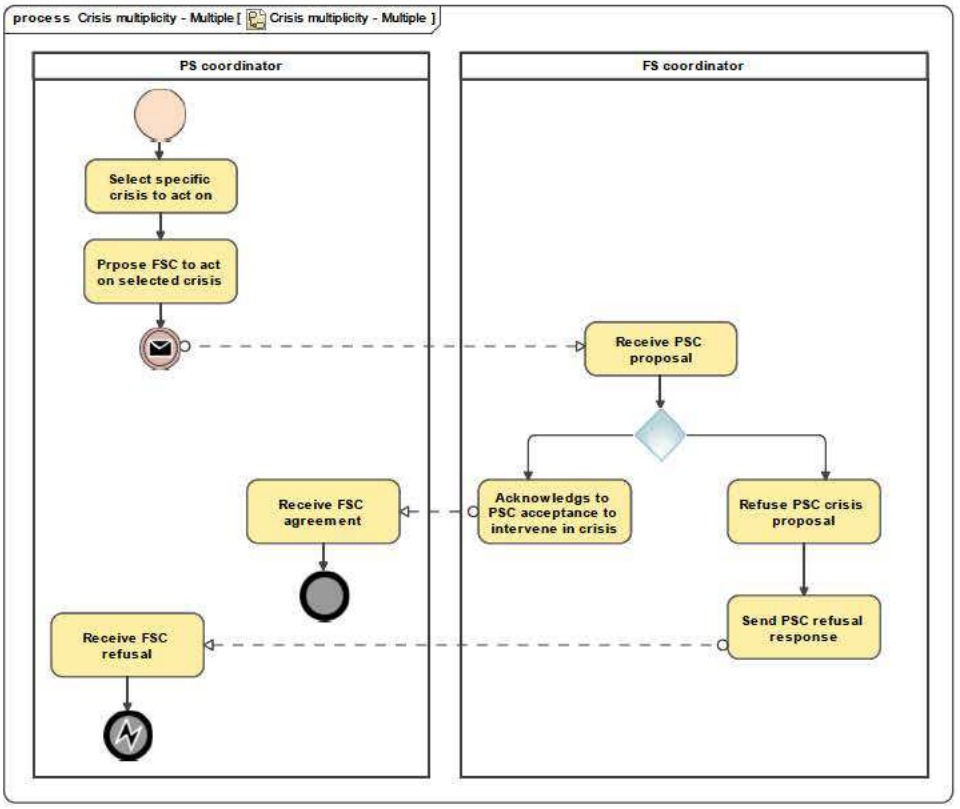


Fig. 6.8: Multiple crisis business process fragment

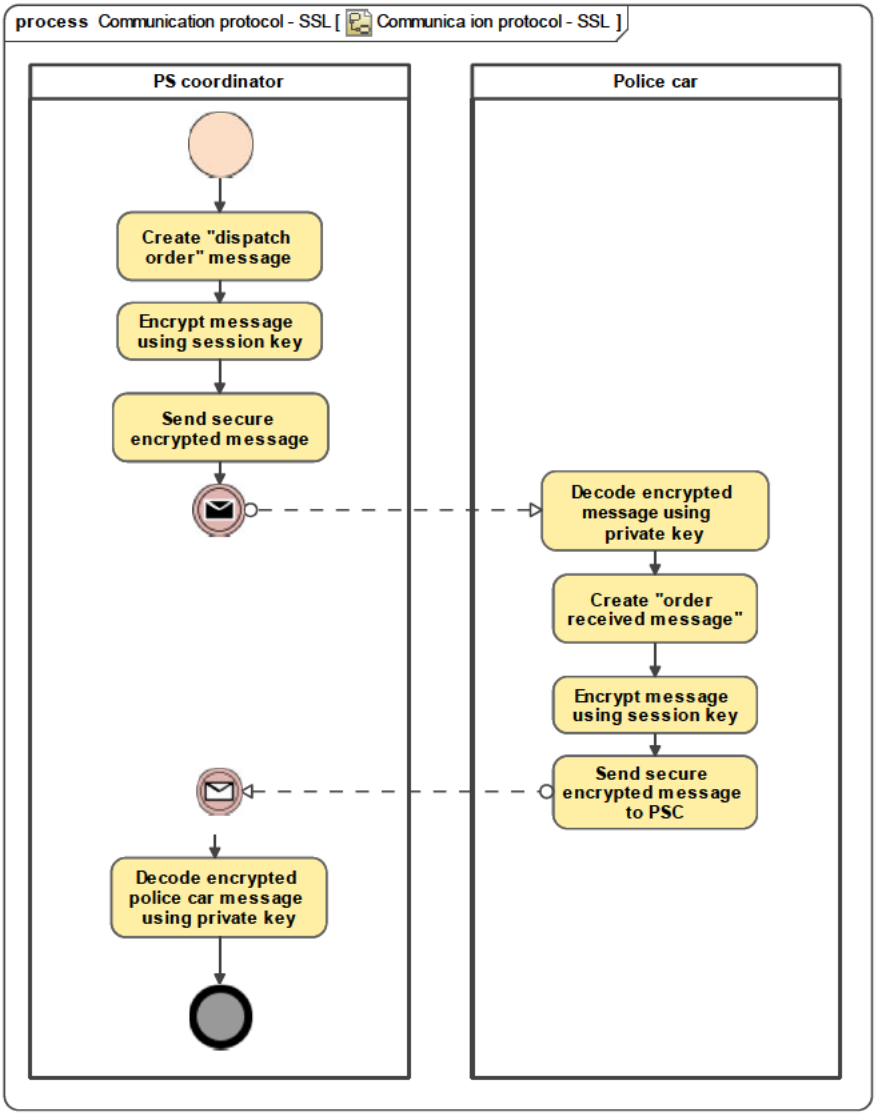


Fig. 6.9: SSL communication protocol for vehicle management business process fragment

on the proposed crisis, it sends its refusal to the PSC, which causes the process to abruptly end with an error. This business process fragment is available in Figure 6.8.

Another example is of the *"Communication protocol by SSL"* business process fragment. It corresponds to the *SSL* feature from the feature diagram, which is a variant of the *Vehicle management communication protocol* variation point and feature. The behaviour modelled describes how the PSC, using a dispatch service, can send a dispatch order to its respective police cars. However, the underlying communication protocol followed for sending the necessary messages follow the SSL protocol. Figure 6.9 graphically depicts this business process fragment.

Finally, the last business process fragment that we present here is called *"Authentication with symmetric encryption"*. It corresponds to the *Symmetric encryption* feature, which is a variant and child of the *Authentication* variation point and feature. It described how, using a symmetric encryption protocol, the PS coordinator can authenticate himself to the

FS coordinator, before sending him his credentials. An authentication authority is required, which generates challenge strings used later on by the PSC for encoding and hashing its authentication credentials. Figure 6.10 graphically presents this business process fragment.

This concludes the presentation of how business process fragments can be created for the bCMS case study. The complete listing of all the business process fragments created for the bCMS case study is available in Annex A.

6.2.3 Verification of business process fragments

Business process fragment verification is a key phase of the methodology. Verification is concerned with determining, in advance, whether a business process model exhibits certain desirable behaviours. In this thesis, we defined the notion of correctness for business process fragments as the summation of two other properties: *structural correctness* and *behavioural correctness*.

Structural correctness mainly focuses on avoiding errors at the structural level of business process fragments. In our case it deals with the correspondence between the model and the CBPF language in which the model is written. It is also concerned with the alignment between the CBPF models and a set of structural properties that any model of the same type must respect.

Structural properties refer to the type and number of elements in a business process fragments and the control flow relations between them. More precisely, to ensure the structural correctness of a business process fragment created using the CBPF language, we need to define a set of adequate *fragment consistency rules* that should be valid for every business process fragment. Thus, in Section 3.4 we proposed a set of well formedness rules defined using OCL directly on the CBPF meta-model. Therefore, all of the models that are created using the CBPF language will be ensured to satisfy these consistency rules. Therefore, in the case of the bCMS case study, all of the business process fragments that were created in the previous section are structurally correct, as they were created with the CBPF language. Thus, there is no need to perform any additional verifications on these business process to know if they are structurally correct. Therefore, following the SPL methodology that we propose, the structural verification step is an automatic one and is guaranteed if we construct the business process fragments using the CBPF language.

However, structural correctness only allows to check that certain structural properties are valid. We also want to perform checks related to the dynamic behaviour of business process fragments. Therefore, in Section 3.4, we defined the notion of *behavioural correctness* which serves to verify the possible behaviours of a business process fragment.

In order to verify the set of behavioural properties defined in Section 3.4, we need first to transform the business process fragment that is being analysed into a hierarchical coloured Petri net. This transformation can be performed by applying the different mapping rules or mapping templates that were proposed as part of the CBPF to HCPN model-to-model transformation.

As an exemplification, we take the *Creation of coordinated route plan* business process fragment. The fragment is described in Figure A.4, available in Appendix A. We want to check that this business process fragments is *behaviourally correct*, which means that it verifies the behavioural properties presented in Section 3.4. To be able to perform these verifications, in a first step we need to transform this business process fragment into a

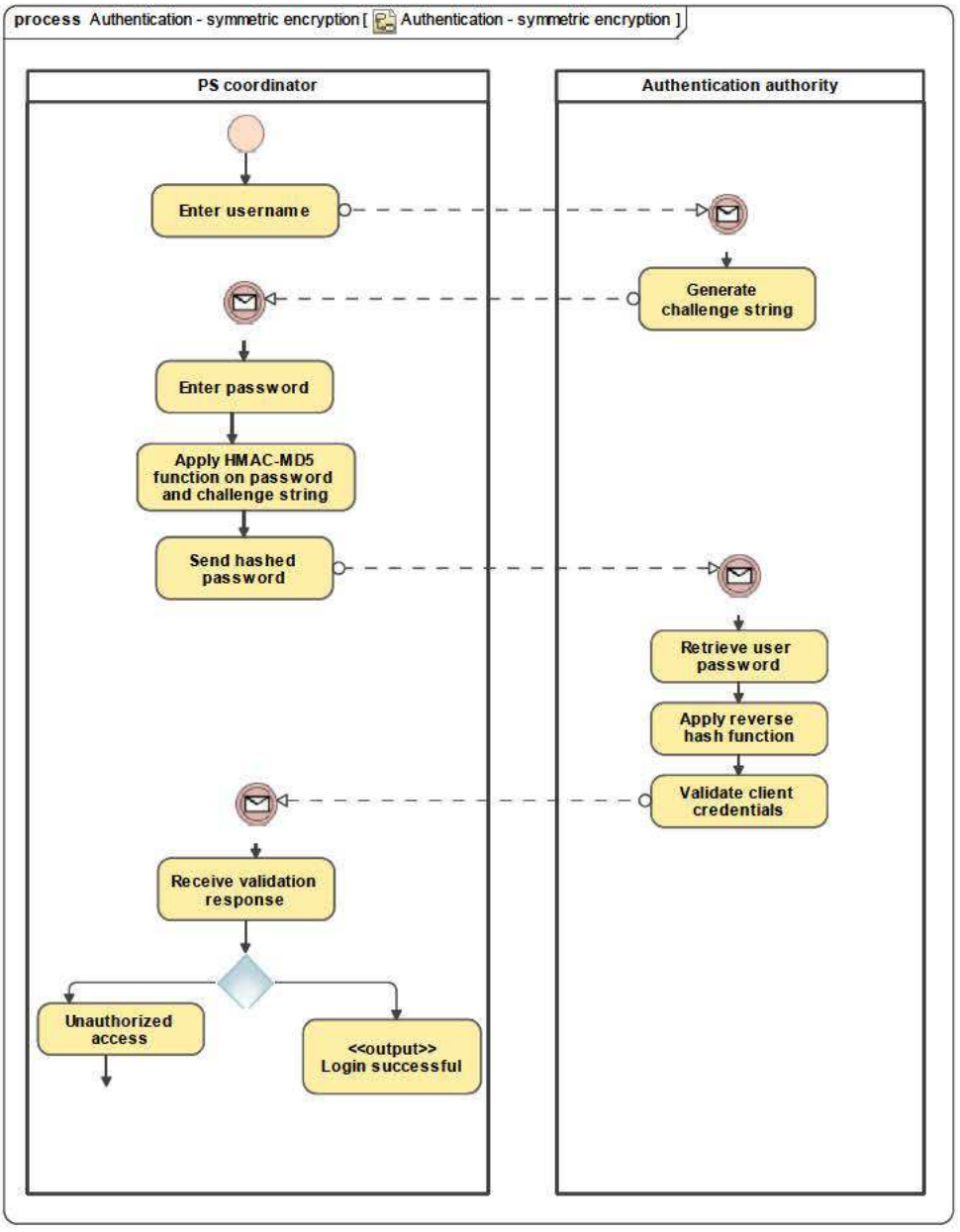


Fig. 6.10: PSC authentication using symmetric encryption business process fragment

Academic Use Only

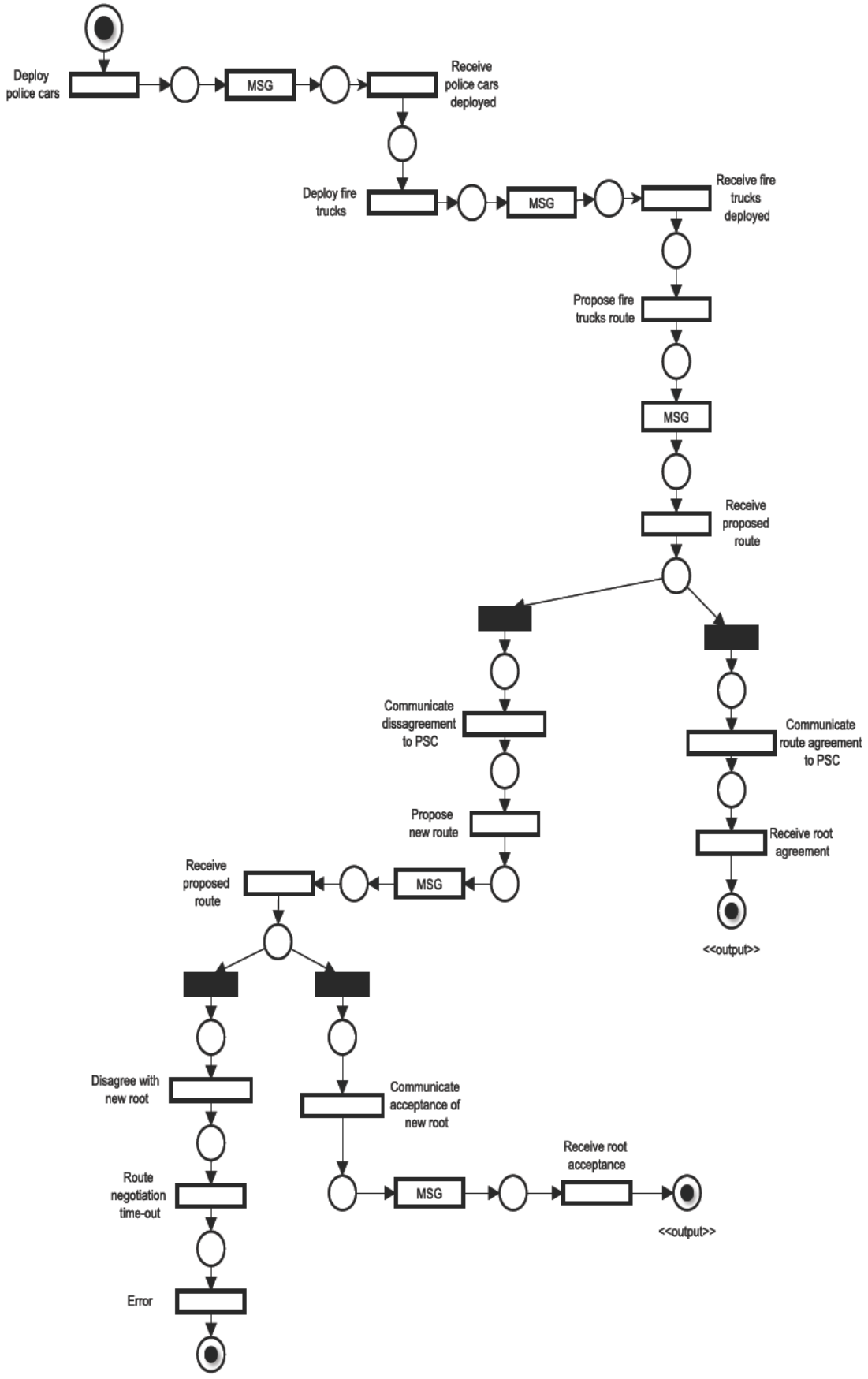


Fig. 6.11: Transforming the *Creation of coordinated route plan* business process fragment into a HCPN

corresponding hierarchical coloured Petri net. Therefore, we apply the proposed mapping templates on the business process fragment under study and obtain its corresponding HCPN model, presented in Figure 6.11.

Once this is done, we can take advantage of the CPN Tool verification capabilities and start by checking whether the general behavioural properties defined in Section 3.4.

- *Reachability of end events:* looking at the original *Creation of coordinated route plan* business process fragment, it can be noticed that it contains three end events: two normal end events and an error end event. Thus, we need to verify that all these three events can be reached from the start event. In practice, we need to apply the $Reachable'(id_{node-start};\ id_{node-end})$ query function on the resulting HCPN model three times. For each application the first parameter will be 1 (id of start node), while the second parameter will take the id of the three end nodes from Figure 6.11. In all three cases, the response of the tool will be YES, followed by a sequence of node ids denoting a possible path from the start node to each of the end nodes.

- *Proper completion of a business process fragment:* in order to verify this property we need to use a combination of home and query functions. We first apply the *ListHomeMarkings*() function. It returns the list of all the home markings of the Petri net. By simply checking the result, we observe that the ids of the end nodes are within this list. Thus, we can proceed and apply the *ListDeadMarkings*() query function. As before, we check that the result contains the ids of the end nodes and see that this is the case. Thus, we can conclude that the property is fulfilled for the business process fragments under analysis.

- *Reachability of composition interfaces:* in the original *Creation of coordinated route plan* business process fragment there are three composition tags applied: one *input composition tag* applied on the start event, and two *output composition tags* applied on each of the two normal end events. Proving that the start event is reachable is trivial. Moreover, proving that the two end events are reachable has already been proven for the first behavioural property that we checked.

- *Absence of dead tasks:* to verify that a business process fragment has no dead tasks, we can simply apply the *ListDeadTIs()* query function. It returns the list of all the dead transitions of the HCPN model under study. Thus, we apply this function on our Petri net and obtain as a result an empty list. This means that there are no dead transitions in the HCPN. Thus, we can conclude that the business process fragment under analysis has no dead tasks.

- *Deadlock-free business process fragment:* we can simply apply the *ListDeadMarkings()* query function on the HCPN model. The result returned by the tool only contains the ids of the end nodes, which are the only dead markings of the net. Thus, we can conclude that the *Creation of coordinated route plan* business process fragment is deadlock-free.

After performing the above-mentioned generic behavioural property verifications, the SPL methodology offers the product line engineer the possibility to perform several fragment specific behavioural verifications. We proposed in Section 3.4 to achieve this by providing the product line engineer with a set of *high-level property templates*, which he can then

tailor and adapt to his particular needs. We apply some of these templates for the *Creation of coordinated route plan* business process fragment and verify some fragment specific properties.

- We want to check that the *Receive proposed route* task of the business process fragment can always be reached from the start event. This specific property can be easily checked by applying the following property template: $Reachable'(id_{node-start};\ Var : id_{node-interested})$. We need to adapt this generic template to our specific request. Thus, we apply the template where the first parameter takes the value 1 (id of the start node), and the second parameter is the id of the place from the net that follows the *Receive proposed route* transition. The query returns TRUE and also provides a list of nodes denoting a possible path from the start place until the *Receive proposed route* transition. Thus, we can conclude that the *Receive proposed route* task can always be reached.
- Another property that we might want to check is that *Propose fire trucks route* task is always executed, in all the possible execution traces. To check this property, we can apply the following property template: $HomeMarking(Var : id_{node-interested})$. We adapt this generic property template for our specific case and will use as parameter for the function the id of the place that follows the *Propose fire trucks route* transition. The function returns true, so the property is fulfilled.
- As another example, we want to check that if task *Receive police cars deployed* is executed, then there exist at least one execution path where task *Receive proposed route* will also be executed. We can use in this case the following property template: $NodesInPath(id_{node-start};\ Var : id_{node})$. We apply it in the following manner: the first parameter is 1; the second parameter is the id of the place that follows the *Receive proposed route* transition. The function returns a list of node ids. We then verify that the id of the place that follows the *Receive police cars deployed* transition is in this list. This is the case, so this property is verified for our business process fragment.

6.2.4 Association of business process fragments to features

During the initial step of the methodology, we used feature models to capture the commonality and the variability of the bCMS product line. The resulting feature model abstracts from concrete feature realizations. However, we need to relate these features to reusable assets describing the solution space. In order to build concrete bCMS products, features have to be realised using software artefacts shared across the product line. For this case study, the core assets of the bCMS product line are the business process fragments already created in one of the previous steps of the methodology.

During this phase of the methodology, we aim at bridging the gap between feature models and the business process fragments of the bCMS product line. Therefore, we define a mapping of features to business process fragments specifying the concrete feature realisations. This mapping was pre-planned and known in advance, as the business process fragments for bCMS were created as concrete implementations for the features from the bCMS feature diagram. This mapping also supports the transition from problem space to solution space in an automated way. It will also allow for the automatic derivation of a product instance based the presence of the features in a variant model that is an instance of a feature model.

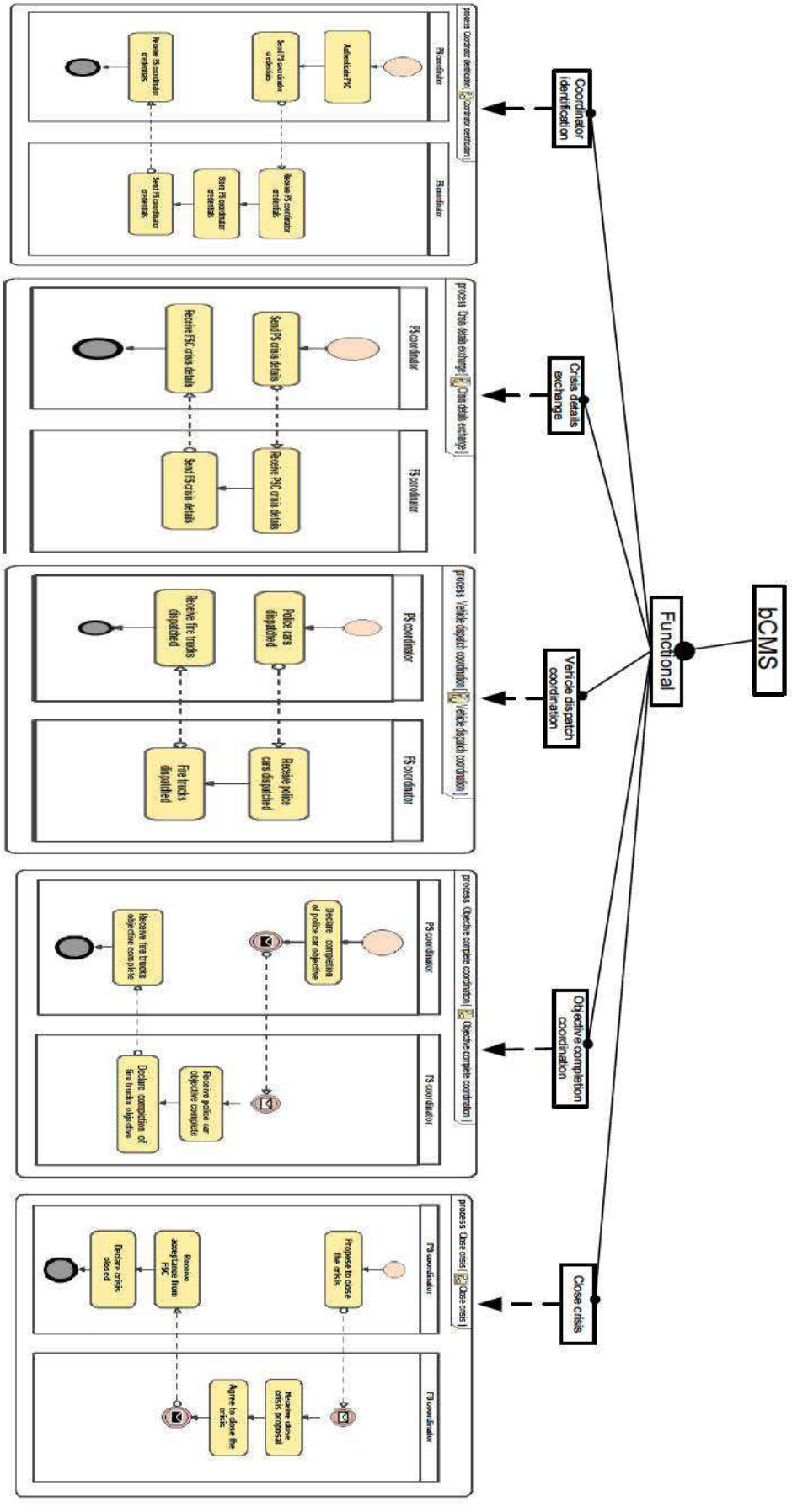


Fig. 6.12: Connecting features to business process fragments for the bCMS case study

As the business process fragments for bCMS were created based on the feature descriptions and their purpose is to be the concrete feature implementations, associating the fragments to the features from the bCMS feature diagram is quite straightforward. When we constructed the business process fragments during the *business process fragment construction step*, for each fragment created we mentioned also the feature to which it corresponds. Also, to facilitate the mapping, the names of the newly created business process fragments were given in such a way as to coincide with those of the features to which they will be associated. An excerpt of this mapping between features and business process fragments for the bCMS example is presented in Figure 6.12.

This concludes the *domain engineering* part of our methodology. In the next sub-sections, we present how concrete bCMS products can be obtained during the *application engineering* phase.

6.2.5 Configuration of the feature diagram

We start by selecting the required features that will be part of a particular bCMS product that we want to derive. The actual feature selection process is based on user requirements and choices, therefore this step of the methodology highly involves the end-user.

The bCMS feature model previously created describes the configuration space of the bCMS product family. It represents a set of configurations, each being a set of features selected from the bCMS feature model according to its semantics. The product line engineer may specify a member of the bCMS product line by selecting the desired features from the feature model within the variability constraints defined by the model. These are instances of the feature diagram and consists of an actual choice of atomic features, matching the requirements imposed by the diagram.

In order to obtain such a feature diagram configuration, we need to select or remove features from the bCMS feature diagram (while taking any constraint into account), in order to reduce the variability that the feature model is depicting. A configuration consists of the features that are selected according to the variability constraints defined by the feature diagram. The outcome of the configuration process will be a concrete configuration which uniquely identifies a product in the bCMS product line. In Section 3.6 we explained in detail how this process is performed and which are the rules that apply when performing a feature diagram configuration.

We want to obtain a bCMS product with a large number of features, in order to increase the complexity of the example and point out the feasibility of our approach. Thus, the bCMS product that we want to obtain should have the following characteristics/features:

- it should be able to perform the basic functionalities defined in the *Functional requirements* section of the bCMS requirements document;
- the product describe the case of a single police station and fire station that manage the crisis;
- the system can handle multiple crises;
- offers a vehicle management functionality that allows both police and fire stations coordinators to send and receive messages to/from their respective vehicles;

- the communication between the police station coordinator and the fire station coordinator will be done using a SOAP communication protocol;
- the underlying communication layer used by the bCMS system is based on HTTP;
- the system supports an authentication mechanism based on symmetric encryption;
- data communication confidentiality is ensured by having encrypted exchange of crisis details between the PSC and FSC.

For the bCMS case study, we start the configuration process by first selecting the *root feature* from the bCMS feature diagram presented in Figure 6.3. The root feature of any feature diagram is the smallest prospective configuration, therefore it has to be selected. We then continue by performing the *core selection:* any *mandatory* child feature of the root feature, and subsequently, any other mandatory child feature connected indirectly to the root feature through mandatory child features, belongs to this core selection. These feature need need to be selected for any product that we want to derive. They define the basic functionalities of any bCSM product. In our case, this leads to the selection of the following features for our bCMS feature diagram configuration: *functional*, *non-functional*, *communication establishment*, *coordinator identification*, *crisis details exchange*, *coordinated route plan creation*, *vehicle dispatch coordination*, *vehicle target arrival coordination*, *objective completion coordination*, *vehicle return coordination*, *close crisis*.

The next step that needs to be taken is to resolve all the variation points defined in the initial bCMS feature diagram. The resolution of the variation points is done based on the specific choices made before, when we stated what characteristic we would like our bCMS product to have. For the bCMS variation points, we perform the following selections:

- *Police and Fire station multiplicity*: variant *"One PS and FS"* is selected. The XOR feature relation automatically excludes the other possible variant *"Multiple PS and FS"*.
- *Vehicle management communication protocol*: variant *"SOAP"* is selected. The other variant, defined as optional, is not selected. Once the feature *Vehicle management communication protocol* is selected, due to the *require* feature dependency present in the bCMS feature diagram, we are also obliged to select one of the variants of feature *Other*, child of feature *Vehicle management*.
- *Vehicles management*: both variants *"PSC send and receive"* and *"FSC send and receive"* are selected. The other variants, defined as optional, are not chosen.
- *Crisis multiplicity*: variant *"Multiple"* is selected. The XOR feature relation automatically excludes the other possible variant *"Single"*.
- *Communication layer*: variant *"HTTP"* is selected.
- *Authentication*: variant *"Symmetric encryption"* is selected. The presence of the XOR feature relation automatically excludes features *"Kerberos"* and *"Mutual authorization based"*. The other possible variants, defined as optional, are not selected.
- *Data communication confidentiality*: variant *"Encrypted"* is selected. The XOR feature relation automatically excludes the other possible variant *"Not encrypted"*.

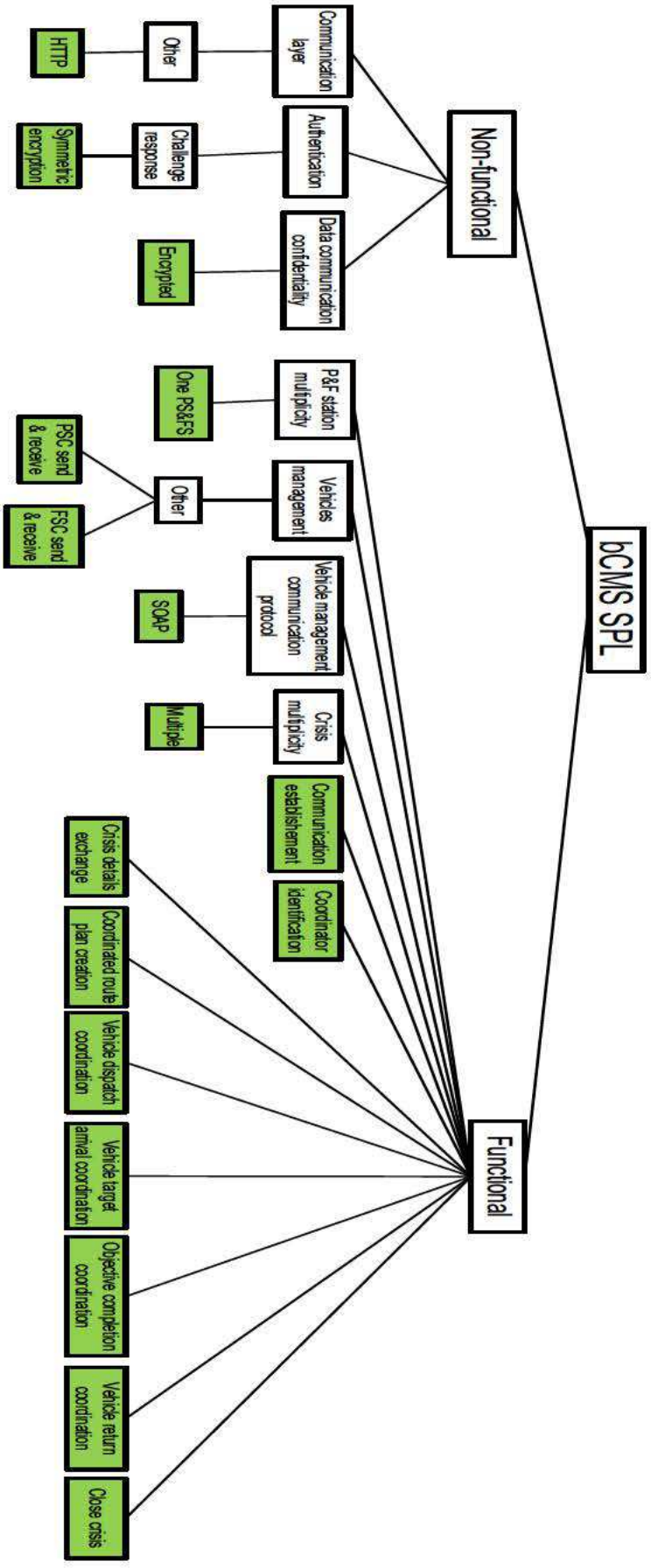


Fig. 6.13: Feature diagram configuration of bCMS product

The final result of the configuration process is presented in Figure 6.13. The diagram corresponds to a specific bCMS product. It can be noticed that there is no more representation of variability in the obtained diagram, as all the variation points of the original bCMS feature diagram have been resolved. Features marked in green in the diagram denote features for which a business process fragment has been created. These features thus each have a business process fragment associated. The other features in the diagram correspond to intermediate features from the original bCMS feature diagram and only serve decomposition and representation purposes.

Once a selection of features has been made and a feature diagram configuration obtained, as in one of the previous steps of the methodology we associated business process fragments to the features, a selection of such business process fragments is also automatically made. For each selected feature, the corresponding business process fragment is also automatically selected. Therefore, the end result of this step of the methodology is a set of business process fragments that correspond to the selected features and denote the functionalities of the bCMS product that we want to derive.

6.2.6 Product derivation specification

The last phase of the methodology is called *product derivation specification*. It takes as input the set of business process fragments resulting from the previous step and transforms them, using a compositional approach, into a proper business process that models the behaviour of a bCMS product.

The business process fragments resulting from the previous step of the methodology need to be composed together. *Composition interfaces* are an essential part of the composition process and thus of the entire product derivation. The business process fragments available at this step of the process have no composition interfaces defined on them. Therefore, during this step of the process, composition interfaces are defined on the entire set of business process fragments corresponding to the bCMS product we are deriving. The annotation is performed iteratively for each business process fragment, until all of them have been annotated.

We start by annotating the business process fragments that correspond to the *core selection of features*. They are those fragments that will actually appear in every bCMS product and define the basic behaviour for all bCMS products. For these business process fragments, most of the annotations performed are on the *start and end events*. That is because we know that there is a logical and functional dependency between these fragments and that the pieces of behaviour each fragment describes need to be composed in succession for obtaining the behaviour of the final bCMS product. Therefore, the following annotations are performed:

- **Communication establishment:** this fragment already has an output composition interface defined at the *Receive FS coordinator call* task. Additionally, we add an *input composition interface at the start event*.

- **Coordinator identification:** for this fragment, we add an *input composition interface at the start event* and also an *output composition interface at the end event*. We also add an *output composition tag on task "Authenticate PSC"*.

- **Crisis details exchange:** for this fragment, we add an *input composition interface at the start event* and also an *output composition interface at the end event*. There will also be *input composition tags* added on all the four tasks of the business process fragment.
- **Coordinated route plan creation:** for this fragment, we add an *input composition interface at the start event*. We also add two *output composition interfaces*, one at each regular end event of the business process fragment.
- **Vehicle dispatch coordination:** for this fragment, we add an *input composition interface at the start event* and also an *output composition interface at the end event*. In addition to this, we add *two input composition interfaces:* one at the *Police cars dispatched* task and the other at the *Fire trucks dispatched* task.
- **Vehicle target arrival coordination:** for this fragment, we add an *input composition interface at the start event* and also an *output composition interface at the end event*.
- **Objective complete coordination:** for this fragment, we add an *input composition interface at the start event* and also an *output composition interface at the end event*.
- **Vehicle return coordination:** for this fragment, we add an *input composition interface at the start event* and also an *output composition interface at the end event*.

We then continue to add annotations on the rest of the fragments. These fragments correspond to specific choices made for the different variation points of the bCMS feature diagram and thus define pieces of behaviour specific to the bCMS product that we are deriving. The following annotations are made:

- **PSC send and receive:** for this fragment, we add an *input composition interface at the start event* and also an *output composition interface at the end event*. We also add an *input composition tag* on task *broadcast order to police cars*.
- **FSC send and receive:** for this fragment, we add an *input composition interface at the start event* and also an *output composition interface at the end event*.
- **Crisis multiplicity - multiple:** for this fragment, we add an *input composition interface at the start event* and also an *output composition interface at the normal end event* of the business process fragment.
- **Communication protocol - SOAP:** for this fragment, we add an *input composition interface at the start event* and also an *output composition interface at the end event*.
- **Communication layer - HTTP:** for this fragment, we only add an *output composition interface at the end event*.
- **Symmetric encryption:** this business process fragment already has a composition interface defined on it from its creation, an *output tag on the "Login successful" task*. We will also add a new *input composition tag at the start event*.
- **Data communication confidentiality - encrypted:** for this fragment, we add an *input composition interface at the start event* and also an *output composition interface at the end event*. There are also *output composition tags* added on the

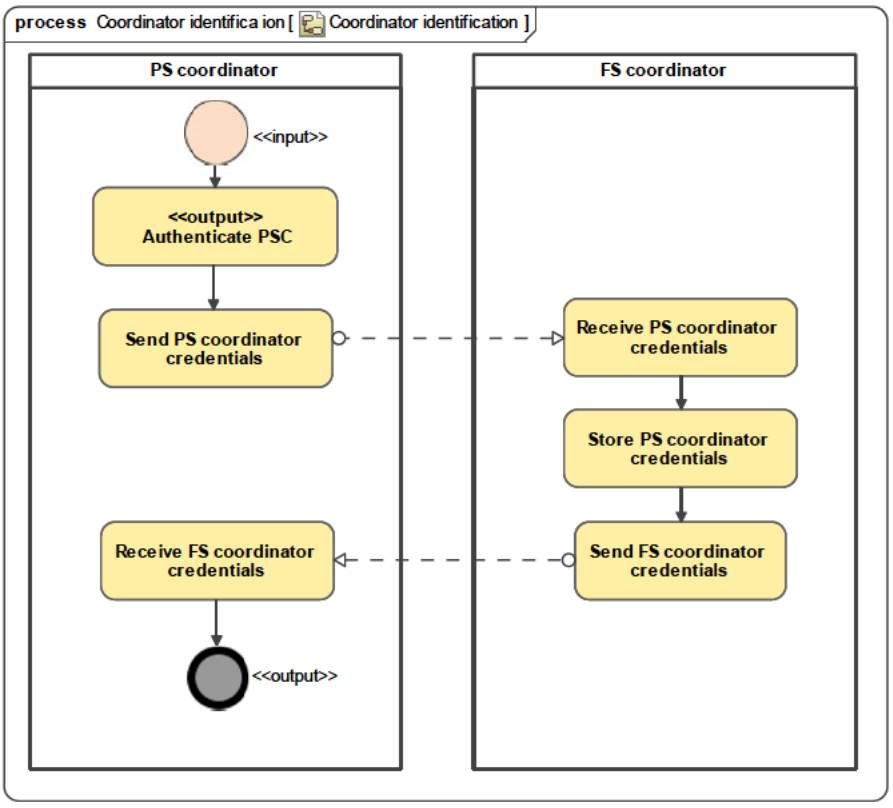


Fig. 6.14: "Coordinator identification" business process fragment after adding composition tags

following tasks of this business process fragment: *"Send PS crisis details"*, *"Receive PS crisis details"*, *"Send FS crisis details"*, *"Receive FS crisis details"*.

As an exemplification, Figure 6.14 shows the business process fragment *"Coordinator identification"* after composition tags have been added on it. Further more, another example is shown if Figure 6.15, where the business process fragment *"Objective complete coordination"* is depicted after the definition of its composition interface.

The next step that needs to be undertaken during the *product derivation specification* process is to *create the composition workflow*. At this moment, several possible orders to compose the annotated business process fragments are possible. To obtain the specific behaviour that characterizes the derived bCMS product, the annotated business process fragments need to be composed in a specific order. The CBPF language proposes the use of a *workflow notation* for specifying this composition order. The composition workflow is specific to each individual bCMS product that we want to derive. It is created by the product line engineer based on the specific composition interfaces of the business process fragments, which will highly restrict the possible orders.

As defined in Section 3.7, the composition workflow consists of the following elements: *fragment place-holders* (black-box representation of the selected business process fragments), *operators* (the composition operators that will be used) and *connectors* (connect fragment place-holders and operators). We present in the following which compositions are made, in which order and using which specific composition operators:

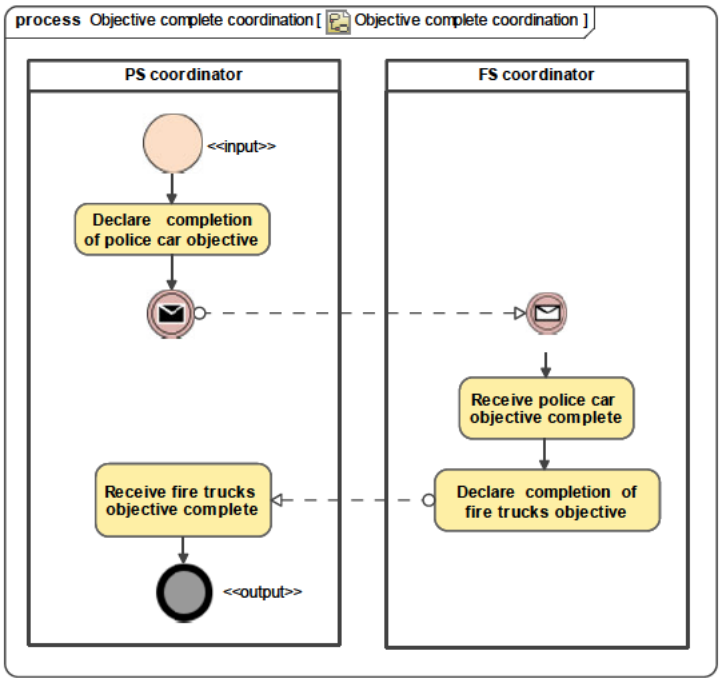


Fig. 6.15: "Objective complete coordination" business process fragment after adding composition tags

Operator	Operand 1	Operand 2	Result	Observations
sequential	*communication layer HTTP*	*communication establishment*	*Result 1*	It can be noticed that fragment *communication layer HTTP* has no input composition interfaces defined on it. This is a good indication that this fragment should be the first one in our composition workflow. The pre-conditions for applying the operator (in terms of available composition interfaces) are satisfied. The obtained fragment has a new composition interface, obtained according to the specific rules of the sequential composition operator defined in Section 4.2.3.
sequential	*Result 1*	*Coordinator identification*	*Result 2*	
refinement	*Result 2*	*Authentication symmetric encryption*	*Result 3*	It is task *Authenticate PSC*, tagged with an *output composition tag*, that will be used for the actual refinement operation.

sequential	*Result 3*	*Crisis multiplicity multiple*	*Result 4*	
synchron.	*Crisis details exchange*	*Data confidentiality - encrypted*	*Result 5*	It can be noticed that these two fragments have four task with the same name: *"Send PS crisis details"*, *"Receive PS crisis details"*, *"Send FS crisis details"*, *"Receive FS crisis details"*. For fragment *Crisis details exchange* they are tagged with *input composition tags*, while for fragment *Data confidentiality - encrypted* they are tagged with *output composition tags*. These tagged activities from the two fragments constitute the *synchronization set* required by the synchronization composition operator.
sequential	*Result 4*	*Result 5*	*Result 6*	
sequential	*Result 6*	*Creation of coordinated route plan*	*Result 7*	
sequential	*Result 7*	*Vehicle dispatch coordination*	*Result 8*	
insertion	*Result 8*	*PSC send and receive*	*Result 9*	An *insert before composition* is performed, at task *Police cars dispatched* which has an input composition tag.
insertion	*Result 9*	*FSC send and receive*	*Result 10*	As before, it as an *insert before composition* that is performed, at task *Fire trucks dispatched* which has an input composition tag.
refinement	*Result 10*	*Communication protocol SOAP*	*Result 11*	The refinement operation is performed at task *Broadcast order to police cars*, belonging to fragment *PSC send and receive* which was already composed.
sequential	*Result 11*	*Vehicle target arrival coordination*	*Result 12*	
sequential	*Objective complete coordination*	*Vehicle return coordination*	*Result 13*	
sequential	*Result 13*	*Close crisis*	*Result 14*	
sequential	*Result 12*	*Result 14*	*Final result*	The result obtained is also the *final result* of our composition process. It denotes a business process that describes the behaviour of the bCMS product that we are deriving using our methodology.

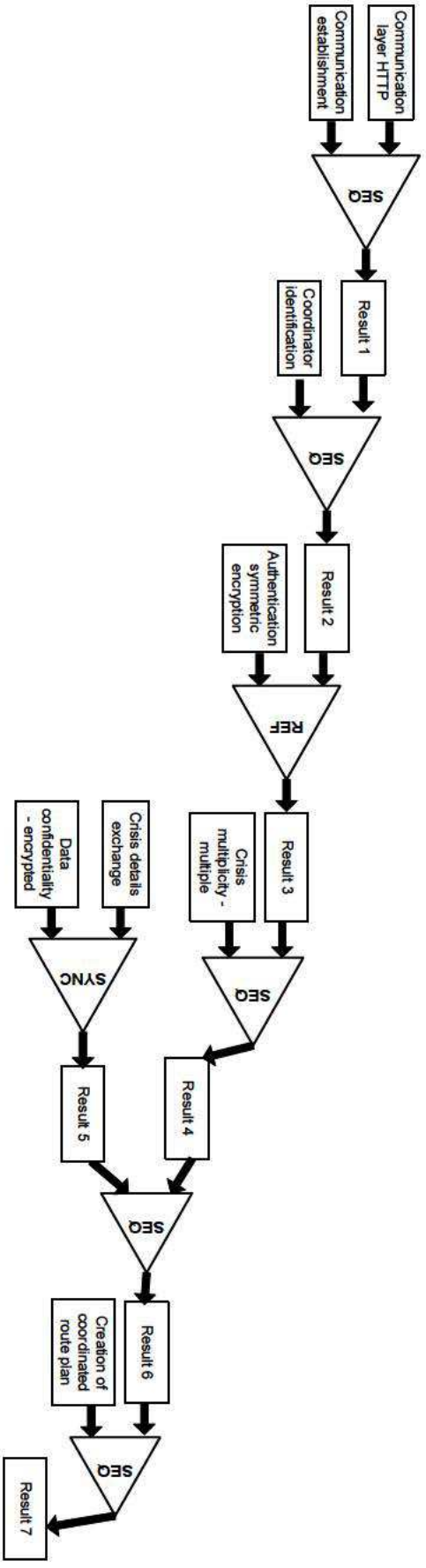


Fig. 6.16: First part of composition workflow for bCMS example

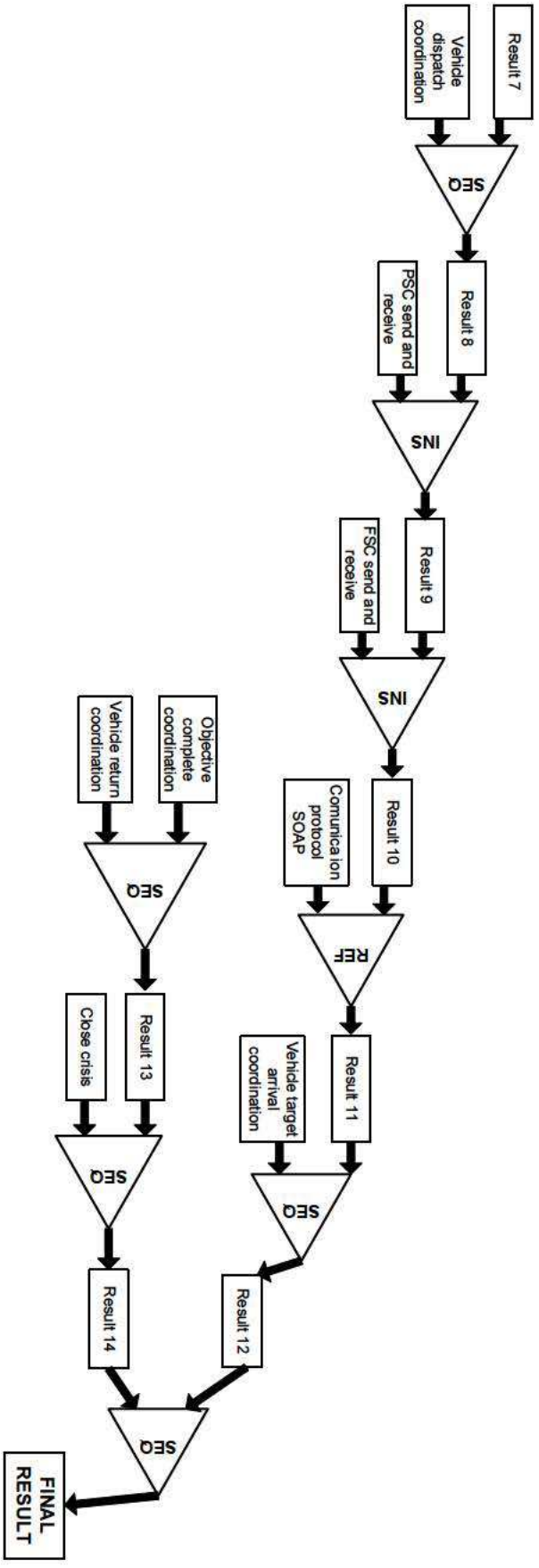


Fig. 6.17: Second part of composition workflow for bCMS example

The composition workflow is created using the CBPF language. The result obtained for the first part is graphically represented in Figure 6.16. The composition workflow that describes the last compositions performed, starting with fragment *Result 7* and leading to the *final result*, is graphically presented in Figure 6.17. Therefore, in order to have the complete composition workflow for the bCMS product that we are deriving using the methodology, we need to concatenate the workflows presented in Figures 6.16 and 6.17. The result gives a complete image of the composition workflow for the bCMS product.

This concludes the presentation of how our methodology can be applied to the bCMS case study.

6.3 Tool support

Throughout Chapter 3 of this thesis, we proposed a new software product line engineering methodology that focuses on the derivation of product behaviour. By applying the proposed methodology, behavioural product models can be produced that belong to the analysis and early design levels of the software development life-cycle. The behavioural models obtained described the business and operational step-by-step workflows of activities/actions performed by the derived product. Then, in Chapter 4 we proposed a new domain specific language called *Composable Business Process Fragments* (CBPF) designed specifically for modelling composable business process fragments. CBPF provides the necessary language support for several steps of our methodology. A model driven approach is followed for creating and specifying the CBPF domain specific language. Finally, throughout Chapter 5 we proposed several types of verifications that can be applied to business process fragments in order to determine their "correctness". These verifications are used during a key step of the proposed SPL methodology. We also use them as we want to ensure that the business process fragments created with the CBPF language during the domain engineering phase are correct.

Therefore, throughout the previous chapters of the thesis we proposed a new SPL engineering methodology and the necessary language support for it. However, in order for this methodology to be easily applicable by product line engineers, it requires also the appropriate level of tool support. Good tool support is one of the key elements for the fast adoption of any new methodology and language. Thus, it is of the utmost importance to provide the product line engineer with a tool that will allow him to practically apply the concepts and ideas proposed by our methodology. Moreover, after designing a domain-specific language like CBPF, the next important task is to determine how to provide the supporting tools for the modelling language. Therefore, throughout this section, we propose the appropriate tool support for our methodology. We start by describing the general requirements that such a tool should fulfil. We then present the general architecture of the proposed tool and discuss in more details the different functionalities it provides.

6.3.1 Tool requirements

The tool that we want to provide needs to facilitate the use of the SPL methodology that we proposed in Chapter 3. Therefore, it needs to support all of the steps of the methodology or as many of them as possible. Thus, the basic requirements that our tool must satisfy are in direct connection with the steps of the methodology. We discuss in the following the set of requirements (features) that the proposed tool should respect and provide:

- *Support for modelling and configuring feature diagrams:* feature diagrams are used in our proposed methodology as a means to capture the commonality and the variability of all the products of the SPL. Moreover, they are used during the product derivation phase for configuring the specific SPL product that we want to derive. Therefore, it is important to that the tool provides support for the modelling of feature models. Moreover, another useful characteristic would be to allow the user to make different configurations of the feature diagram, which would be a great asset for the product derivation phase.

- *Support for creating CBPF models:* the CBPF domain specific language was presented in Chapter 4 as our language support proposal for the methodology. It allows the modelling of composable business process fragments. It also allows to define composition interfaces for these business process fragments. Moreover, using a set of OCL constraints, all models created with the CBPF language are structurally correct, as presented in Chapter 4. Therefore, our tool should also allow the product line engineer or any other user to create such CBPF models.

- *Support for verifying business process fragments:* business process fragment verification is one of the key steps of our methodology. The verification of structural correctness is ensured by defining a set of well-formedness rules on the CBPF language meta-models. However, for the verification of *behavioural correctness*, we perform a transformation of CBPF models in HCPN models and perform different behavioural verifications at this level. Therefore, another desirable feature of the tool would be to allow the verification of behavioural correctness of CBPF models by means of Petri net verifications.

- *Support for composing business process fragments:* during the product derivation phase of the methodology, we derive the behavioural representation of an SPL product using a compositional approach. The business process fragments selected by the user will be composed, following a specific composition workflow, in order to obtain the end business process describing the product. Thus, the tool should have the following two features: allow the composition of business process fragments and facilitate the creation and parsing of the composition workflow.

With this requirements in mind, in the following we propose a tool that supports our methodology. We start by presenting the general architecture of the tool.

6.3.2 General architecture of the tool

The tool chain (suite) that we propose is called *SPLIT*. The name comes from the research project in the context of which this thesis took place. The SPLIT tool suite provides a practical implementation of the proposed methodology. The tool is Eclipse-based [1]. Eclipse is a multi-language software development environment comprising an integrated development environment (IDE) and an extensible plug-in system. It is written mostly in Java. It can be used to develop applications in Java and, by means of various plug-ins, other programming languages.

The Eclipse Platform uses *plug-ins* to provide all functionality within and on top of the runtime system, in contrast to some other applications, in which functionality is hard

[1] http://www.eclipse.org

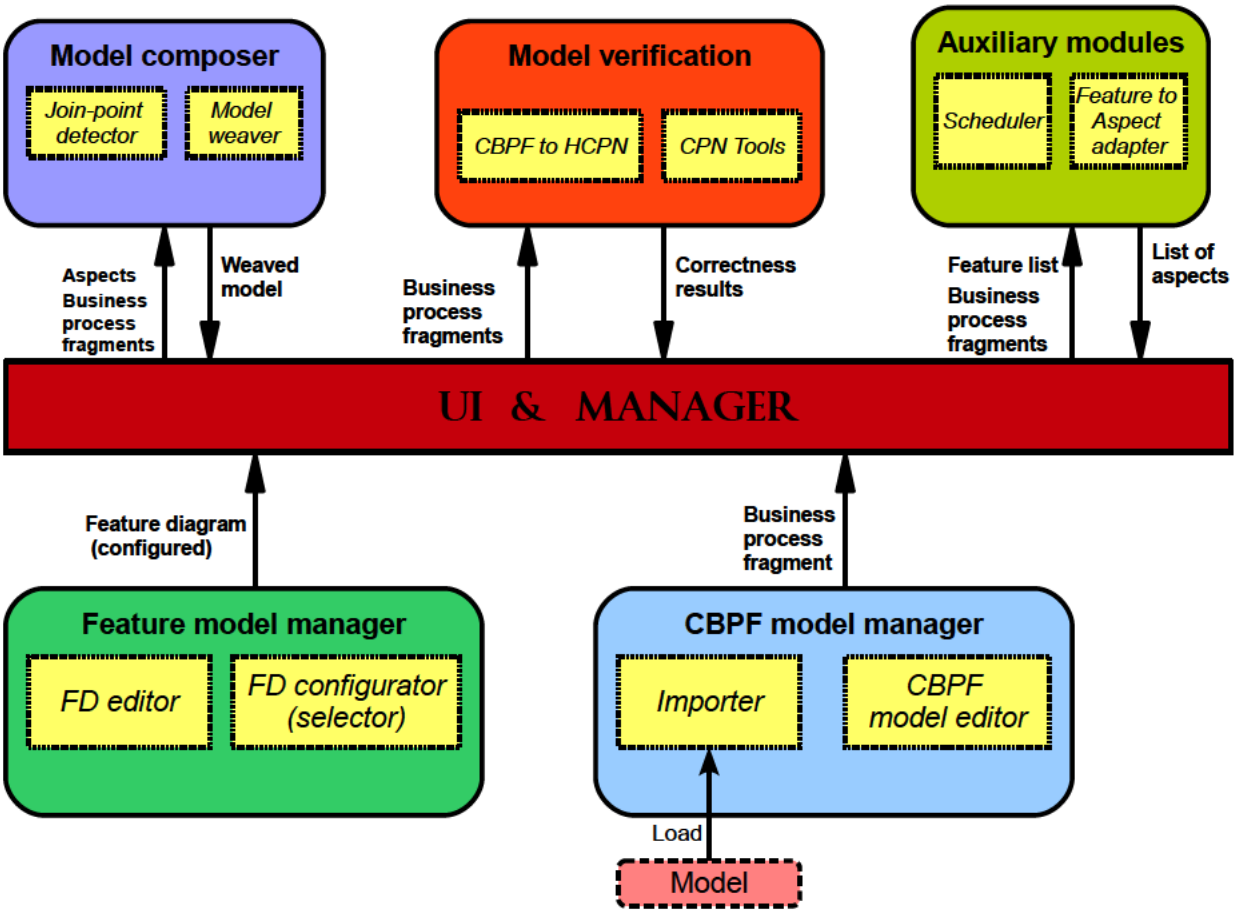


Fig. 6.18: General architecture of SPLIT tool

coded. The Eclipse IDE allows the developer to extend the IDE functionality via plug-ins [2], which are Eclipse software components. The plug-in architecture supports writing any desired extension to the environment. With the exception of a small run-time kernel, everything in Eclipse is a plug-in. Plug-in development consists of developing the code following the "rules" of Eclipse, and so we have the obligation and privilege to use features available in the Eclipse platform.

Therefore, the *SPLIT tool suite* has been developed as a *set of Eclipse plug-ins* which are meant to be integrated as a single tool that is capable of fulfilling the requirements presented in the previous sub-section. This is to facilitate integration with already existing tools, make it distributable through Eclipse repositories and as an eclipse plug-in it facilitates its use in multiple OS's.

Having this in mind a modular architecture was defined, to facilitate plugin-development and development iterations, by distributing different tasks of the tool to different modules of the architecture. Following this modular development approach approach, we can also facilitate development and subsequent research as well as maintenance and future testing and experimentation. This allows the swapping of strategies and approaches, while maintaining a fully functional tool, and in this manner quantitatively and qualitatively compare

[2] http://www.eclipse.org/pde/

multiple implementation options for the same module section.

The general architecture of the tool is presented in Figure 6.18. The tool is highly modular, as we can notice the clear separation into modules. Each module provides a separate functionality, mainly corresponding to the tool requirements described previously. The following core modules are proposed:

- **UI and Manager:** the overall control of all the modules and thus of the functionalities provided by the SPLIT tool chain is handled by the *UI and Manager* component. Its overall goal is to serves as a bus between all the other modules. In practice what this implies is that each module with the exception of the Manager is unaware of the other module's interface, and because of this they are completely independent, facilitating modular development. The manager receives different data from all the other modules and forwards it as input to other modules that need it. Moreover, to provide a greater control of the user interaction and to allow this interaction to better evolve with the remaining tool's development, these interactions are specified in this module. This allows to expand on the user experience without interfering with the functional code of the tool and on the other hand as the tool's inner functioning evolves, the user interaction can remain stable.

- **Feature model manager:** this core module of the SPLIT tool suite provides basic feature modelling and configuration facilities. As it can be seen from Figure 6.18, the module regroups two separate functionalities: modelling new feature diagrams using the dedicated *feature model editor* and, starting form a feature diagram, create different possible and valid configurations for it using the *feature diagram configuration* component.

- **CBPF model manager:** this module of the SPLIT tool chain deals with *business process fragments*. In Chapter 4 we proposed the domain specific language *CBPF* that allows the modelling of composable business process fragments. Therefore, this module allows the user to obtain business process fragments in two ways: create a new business process fragments from scratch using the *CBPF model editor* component; reuse an already existing business process fragment by loading it using the *importer* component.

- **Model composer:** this module deals with the composition of business process fragments for obtaining the end result of the proposed SPL methodology. In the SPLIT tool, the business process fragment composition is implemented using *aspect model weaving* techniques and procedures. Model weaving, being one of many forms of model manipulation, focuses in the combination of two or more models into a singular model. In this approach, we structure the weaving, by classifying the models into two types: the base models, of which we only use one at any given weaving, are normal models; the second type are the models that will be woven into the base model. To be able to weave these models into the base model, we also need to specify a pattern of where this weaving can occur in the base models. To this pattern we call pointcut, and for every model to be woven we have one pointcut. The weaving itself takes place, by detecting the pointcut pattern within the base model. This occurs as an adjacency preserving bijection mapping between the pointcut and a sub-section of the base model and it is denoted as an isomorphism. Therefore, this modules provides two components with interdependent functionality: the *join-point detector* performs the detection of the pointcut within the base model; then, based on these

results, the *weaver* component will perform the actual composition of the base model and the aspect.

- **Model verification:** this modules of the SPLIT tool suite deals with the verification of business process fragments, and in particular with the verification of behavioural correctness. Thus, the *CBPF to HCPN* component implements the model transformation from business process fragments on high level Petri nets. Once this is done, we can take advantage of the already existing Petri net verification tool called *CPN Tools*. It is at this level that all the behavioural verifications proposed in Chapter 6 are performed.

- **Auxiliary module:** as its name states, this module will provide extra functionality required and used by the previously presented modules. For example, the business process fragments created using the *CBPF model manager* modules need to be transformed into *aspects* so that they can be used by the *model composer* for the weaving. This transformation is performed by the *Fragment to Aspect adapter* component. Moreover, during the application engineering phase of the methodology, there is the need to create a composition workflow that gives the order in which the compositions (weavings) will be performed. Moreover, the composition workflow needs to be parsed (interpreted), as this information is required by the *weaver* component. This functionality is provided by the *scheduler* component.

Now that the general architecture of the SPLIT tool suite has been presented and the main modules introduced, we can explain how the tool can be actually used. In Figure 6.19 we present the overall usage workflow for the tool. It describes how the different modules and components are actually used, which are their inputs and outputs. Moreover, we can see how the tool can be used for actually supporting the SPL methodology that we proposed in this thesis. This usage workflow consists of several steps, which are also shown in Figure 6.19:

- *Step 1:* the user (product line engineer) creates a new feature diagram of the product line using the *FD editor*.

- *Step 2:* for the features in the previously created FD, the product line engineer constructs new business process fragments. This can be done in two ways: create new business process fragments using the *CBPF model editor*; reuse an existing fragment and load it using the *Importer*. The obtained business process fragments are sent to the *Manager* module.

- *Step 3:* from the previously created business process fragments, one of them is loaded from the *Manager* into the *CBPF to HCPN* component.

- *Step 4:* the fragment previously loaded into the *CBPF to HCPN* component is transformed into a corresponding high level Petri net. Te result is then sent back to the *Manager*.

  Steps 3 and 4 are repeated until all the business process fragments are transformed into HCPN models.

- *Step 5:* one by one, the obtained HCPN models are loaded from the *Manager* into the *CPN Tool*.

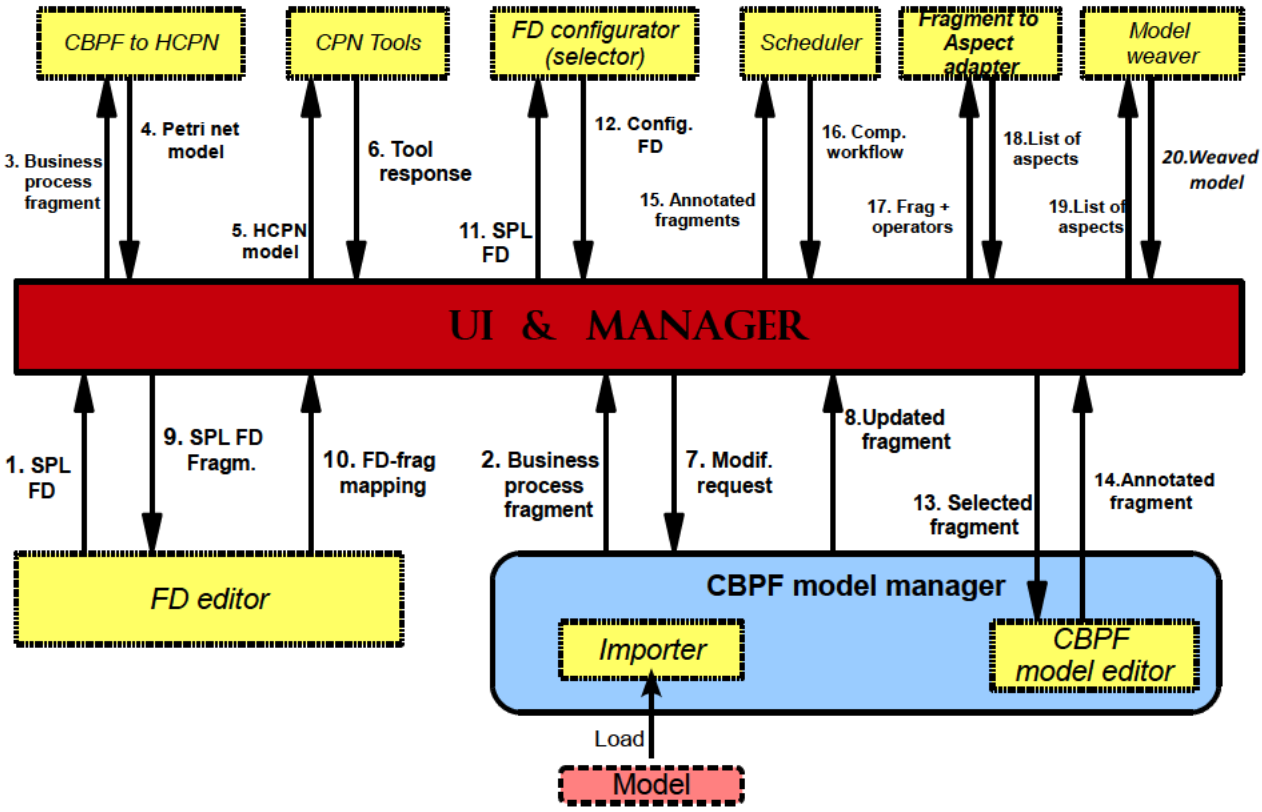


Fig. 6.19: Usage of the SPLIT tool

- *Step 6:* in the *CPN Tool*, the verification of behavioural properties is performed. The tool provides a feedback telling us if the verified properties are true of false.
- *Step 7:* in case some errors have been detected by the *CPN Tool*, then the fragments in cause need to be sent back to the *CBPF model manager* module.
- *Step 8:* the incorrect fragments can be modifies using the *CBPF model editor* and the errors detected can be corrected. After the necessary updates, the fragments are resent to the *Manager*.
- *Step 9:* the SPL feature diagram is loaded into the *FD editor* module.
- *Step 10:* the product line engineer creates a mapping between features and business process fragments. In the *FD editor*, a business process fragment is added into the contained of each feature. In this way, the desired mapping is made.
- *Step 11:* the feature diagram of the SPL is loaded into the *FD configurator*.
- *Step 12:* based on the choices made by the user and on his selection, a configuration of the feature diagram is created. This configuration is stored in the *Manager*. Also, a the list of the business process fragments corresponding to the selected features is also sent to the *Manager*.
- *Step 13:* the business process fragments from the selected list are sent one by one to the *CBPF model editor*.

- *Step 14:* in the CBPG model editor, the product line engineer can annotate the selected fragments with composition tags, creating thus the composition interface for these business process fragments.
- *Step 15:* the annotated fragments are loaded into the *Scheduler*.
- *Step 16:* the composition workflow is created in the *Scheduler*. A parsed version of this workflow is sent back to the manager.
- *Step 17:* the parsed composition workflow (list of business process fragments and composition operators used) is loaded into the *Fragment to Aspect adapter* module.
- *Step 18:* based on the received information, a base model is selected amongst the business process fragments. Then, the other fragments are transformed, one by one, according to the composition order and the composition operator applied, into aspects by the *Fragment to Aspect adapter* module.
- *Step 19:* the base model and the first aspect are loaded into the weaver, then the weaving is applied. The resulting model becomes the new base. The next aspect is loaded and woven. This process is repeated until all the aspects have been woven.
- *Step 20:* the business process obtained at the end of the weaving process is the end result.

6.3.3 Modules of the tool

We have seen that the SPLIT tool suite is highly modular. An overall look at the modules used by the tool was given in the previous sub-section. A more thorough presentation of the different modules use and their functionality is presented in the following.

Feature model manager:

This core module of the SPLIT tool suite provides basic feature modelling and configuration facilities. The feature diagram tool suite is used to manage variability between a family of products. It was created at the *Triskell research team* [3] from Rennes, one of the partners of the SPLIT project [4], in the context of which this thesis took place. The feature diagram tool suite is composed of the following modules :

- *Feature Diagram Editor*;
- *Feature selection engine.*

We choose to create a graphical feature diagram editor as an *Eclipse plugin*. This graphical editor must take into account the following elements : features, decomposition edge such as and, or, xor, card, attributes which can permit to associate metadata on a feature to facilitate the selection of its children. Moreover, the plugin provides a direct mapping between elements of the feature model and elements of the base model(s), implemented as the addition of model elements from this base model(s) into the features. These model elements can be any element stored into a model based on EMF (and ecore). In addition to

[3] http://www.irisa.fr/triskell
[4] http://wiki.lassy.uni.lu/Projects/SPLIT

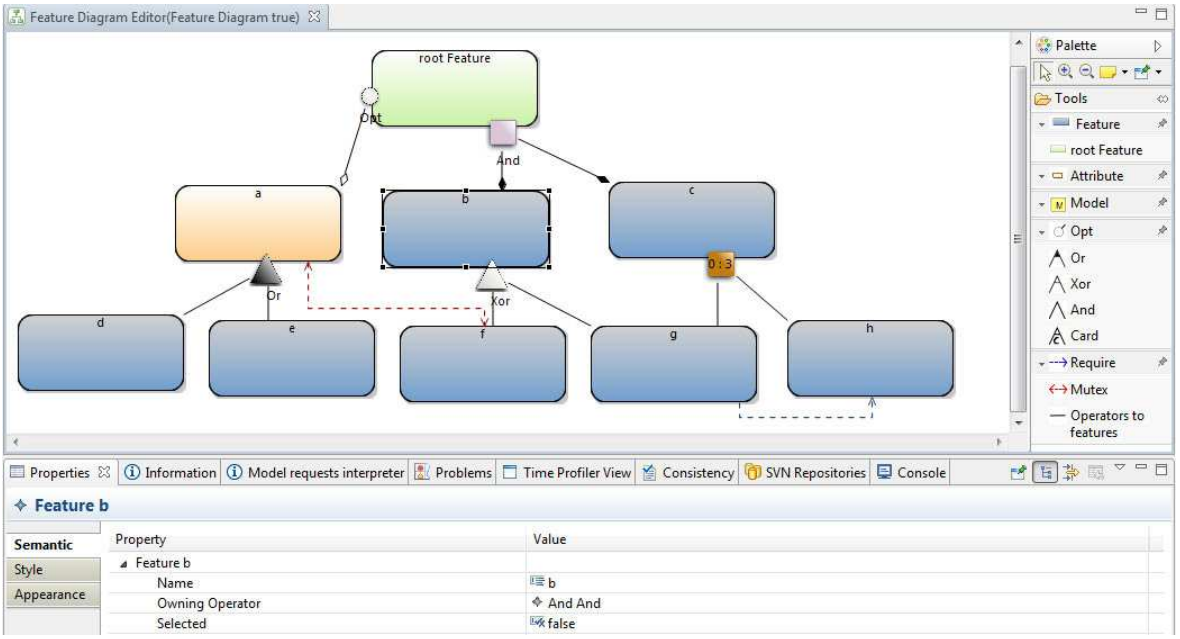


Fig. 6.20: Screenshot of the Feature Diagram Editor

the feature diagram editor, a constraints plugin was developed in order to help user to create valid feature models. This plugin is written using Praxis rules. The rules implemented are the ones described in Section 3.2.

As there exist several feature diagram notations descried in the research literature, we have selected a specific one to be implemented by the feature diagram editor. The used graphical notation is similar to the *FORM notation*, except that we add an *OR operator* (represented with a dark mathematical angle), a *CARD operator* (represented with a white mathematical angle and its bounds) and *Require* and *Mutex* constraints (represented by dashed arrows, one for require and two for mutex). The exact notation used and the feature diagram meta-model used by the tool are the ones presented in Section 3.2.

The technology used to develop this Feature Diagram Editor is *Obeo Designer* [5] provided by the Obeo society. Obeo Designer is a commercial tool which permits to create easily Eclipse-integrated graphical editor for any DSML (GMF-like editor). Obeo Desiger permits to develop a DSL graphical editor on interpretation mode without generating code like in GMF (Graphical modelling Framework). However, some graphical notation cannot be obtained with simplicity. That is why some changes were made between the "conceptual" graphical notation and the real notation on the tool. So, the dark or white mathematical angles respectively for or or xor operator are replaced respectively by a dark or a white triangle. It is the same for the card operator represented with a square. The second tool used for the feature diagram editor is Praxis for expressing constraints on feature diagrams. Praxis [**?**] is an integrated Eclipse tool developed by UMPC to check inconsistencies not only on a single model but also on two distinct models. It is based on Prolog. Praxis was used to create constraints on Feature Diagram Editor.

This feature diagram editor is directly integrated on Eclipse. A screenshot of the tool is presented in Figure 6.20. A demonstration of this tool is available in [INR10]. To add

[5] http://www.obeo.fr/pages/ obeo-designer/fr

graphical element on the feature diagram, the user simply needs to click on the desired tool presented on the right of Figure 6.20 and drag and drop it in the feature diagram canvas. All the classical feature diagram elements described in Section 3.2 can be created using the palette.

A core functionality provided by the Feature Diagram Editor tool is to add domain model elements into features. In most cases, domain model elements are added using drag and drop on a feature. In a first step, we need to add the domain model in the current session. The domain model element(s) is added as an existing resource. The user simply needs to click on the concerned model element and drag it into the feature. It will be added into the container associated to the feature. However, as domain model elements are directly referenced by the features, to add Domain model element on a feature we can alternatively simply right click on the *Feature* $\rightarrow$ *AddDomainModelElement* in the tool. A wizard appears. Click on load button to select a domain model element model. The next wizard page permits to select the desired Domain Model elements and add it into the feature. A special wizard permits to select and associate to a feature any model file based on EMF by clicking on Load and search the file.

A second part of the *Feature model manager* module is the *Features Diagram Configurator (Selection Engine)*. It lets the designer choose which features are required for a specific product. The first version created was a textual interface implemented in Kermeta [6] that traverses the feature model, asks to the designer the feature to select between the children of a given feature and populates a feature selection model (called resolution model). To check this feature selection we will check the resolution model. The tool permits also to select automatically require features and deselect features mutually exclusives with other features ever selected. As a development of the pool, a form model was added that permits to generate automatically a customized user interface dialogue according to the choices proposed to the designer by the feature model.

CBPF model editor:

Business process fragments are the core assets used by our SPL methodology. The CBPF language presented in Chapter 4 provides the necessary language support for modelling new business process fragments. However, we need also the tool support for creating such business process fragments. This is the role of the *CBPF model editor* module of the SPLIT tool suite.

The CBPF language is based on the BPMN language, the standard for modelling business processes, and shares a lot of common elements with it. Therefore, as there are several tools that offer support for modelling business process fragments, we decided to extend and adapt one of these tools in order to fit the needs of the CBPF language. We have selected to base our module on the *Eclipse BPMN Modeler* [7], which is an open source project under the Service Oriented Architecture (SOA) Container Project of Eclipse.

The *BPMN Modeler* provides a graphical modelling tool which allows creation and editing of BPMN (Business Process Modeling Notation) diagrams. The tool is built on Eclipse Graphiti and uses the BPMN 2.0 EMF meta model developed within the Eclipse Model Development Tools (MDT) project. This meta model is compatible with the BPMN 2.0 specification proposed by the Object Management Group (OMG). The following features are in scope for the BPMN Modeler project:

[6] http://www.kermeta.org/
[7] http://eclipse.org/proposals/soa.bpmn2-modeler/

- Basic BPMN 2.0-compliant file creation and editing capabilities;
- Process Modeling, Process Execution and Choreography Modeling Conformance as defined section 2 of the specification;
- Plug-in extension points that allow the editor to be customized for specific applications;
- Deployment of BPMN resources to a suitable runtime;
- Simulation and debug support of business processes.

The specific intent of this project is to provide an intuitive modelling tool for the business analyst, which conforms to well-established Eclipse user interface design practices. The BPMN 2.0 Modeler provides visual, graphical editing and creation of BPMN 2.0-compliant files with support for both the BPMN domain as well as the Diagram Interchange models. Currently the editor is functional and can consume and produce valid BPMN 2.0 model files.

The BPMN Modeler fully leverages the Eclipse Graphical Modeling Framework (GMF) that provides components in order to develop editors based on EMF (Eclipse Modeling Framework) and GEF (Graphical Editing Framework). Following GMF best practices, a flexible and extensible domain model for BPMN has been first created using EMF followed by a corresponding Graphical Model. The GMF generator model was later used to map those 2 models to generate the corresponding BPMN diagram Editor.

However, the editor cannot be used as is and some adaptations are needed. The graphical interface of the Eclipse BPMN Editor offers a palette that enables users to create different graphical BPMN elements and, by drag and drop, add the on the current BPMN diagram that is being modelled. In order to offer support for creating CBPF diagrams, we propose to personalize the palette of the Eclipse BPMN editor. In this way, we can remove shapes from the palette that exist in BPMN but do not exist in CBPF. Moreover, we can also add new elements to the palette and shapes for representing the elements newly introduced by CBPF.

In order to make these changes to the Eclipse BPMN Editor, we took advantage of the extension-point mechanism available. This mechanism defines a standard way of adapting and tailoring the editor to specific needs. One of the main features of the editor is to provide such plug-in extension points that allow the editor to be customized for specific applications. First of all, we need to hide some of the BPMN elements from the palette. In this simple way, we can ensure that only elemnts that exist in CBPF can be added in the diagram.

Secondly, we need to be able to create and display the new elements introduced by CBPF. Following the explanations provided in [8], we identified how to add to the BPMN diagram custom data and also how to display and interact with that custom data. Every element of the domain model of the stp.bpmn modeler is an EMF *EModelElement*. This means that it can be attached arbitrary annotations. By default the *EAnnotations* added to the BPMN objects are displayed in the properties view inside the 'BPMN' tabulation. Once the model has annotations it is possible to display in the diagram that the model element associated to a particular shape or connection has been annotated. In order to do this, an extension

[8] http://wiki.eclipse.org/STP/BPMN$_{C}$ $omponent/STP_{B}$ PMN_{P} $resentation_{(}$ $Part_{3}$)

point called *org.eclipse.stp.bpmn.diagram.EAnnotationDecorator* has been implemented in the Eclipse BPMN Modeler. By adapting this extension point, we can add the concepts of *composition tag* and *composition interface* and be able to display them in the diagrams.

CBPF to HCPN transformation:

In order to perform the behavioural verification of business process fragments and to have access to the different verification capabilities provided by CPN Tools, presented previously, business process fragments need to be transformed into hierarchical coloured Petri nets. This activity is performed by the *CBPF to HCPN* modules, which implements a model-to-model tranaformation between the CBPF and HCPN languages using model transformation approaches.

The transformation implements the mapping between CBPF and HCPN which was already defined in Section 4.4.2. The model transformation we proposed was described as a series of *mapping rules* or *mapping templates* that translate the elements defined in the abstract syntax of the CBPF language into equivalent constructs in HCPN. As CBPF is much bigger than HCPN in terms of size and number of elements, the mapping will usually not be 1-to-1, but in most cases a CBPF language element will be translated into an equivalent *set of HCPN elements* (a HCPN construct). The mapping templates proposed range from simple 1-to-1 ones, to more complicated.

The model-to-model transformation is defined using ATL (ATL Transformation Language) [9]. ATL is a model transformation language and toolkit. In the field of Model-Driven Engineering (MDE), ATL provides ways to produce a set of target models from a set of source models. Developed on top of the Eclipse platform, the ATL Integrated Environnement (IDE) provides a number of standard development tools (syntax highlighting, debugger, etc.) that aims to ease development of ATL transformations.

ATL provides a way to produce a number of target models from a set of source models. An ATL transformation program is composed of rules that define how source model elements are matched and navigated to create and initialize the elements of the target models.In other word, ATL introduces a set of concepts that make it possible to describe model transformations.

The ATL language is a hybrid of declarative and imperative programming. The preferred style of transformation writing is the declarative one: it enables to simply express mappings between the source and target model elements. However, ATL also provides imperative constructs in order to ease the specification of mappings that can hardly be expressed declaratively. An ATL transformation program is composed of rules that define how source model elements are matched and navigated to create and initialize the elements of the target models. Besides basic model transformations, ATL defines an additional model querying facility that enables to specify requests onto models. ATL also allows code factorization through the definition of ATL libraries.

Some simple example of mapping rules defined with ATL for the CBPF to HCPN model transformation are presented in the following.

Figure 6.21 presents the ATL rule that maps the *Composable business process fragment* objects into *Petri Net* objects. A matched rule enables to match some of the model elements of a source model, and to generate from them a number of distinct target model elements. It can be noticed that each rule has a unique name. We then need to define the

[9] http://www.eclipse.org/atl/

```
1  rule ComposableBusinessProcessFragment2PetriNet {
2          from
3                  cbpf : MM-CBPF ! Composable_business_process_fragment
4          to
5                  pn : MM-HCPN ! Petri_Net (
6                          name <- cbpf.title
7                  )
8  }
```

Fig. 6.21: CBPF to HCPN tranformation using ATL: mapping the root elements

```
1  rule CBPFObject2PetriNetElement {
2          from
3                  Cobj : MM-CBPF ! CBPF Object
4          to
5                  PNelem : MM-HCPN ! PetriNet element (
6                 id <- Cobj.id
7                          name <- Cobj.name
8                  )
9  }
```

Fig. 6.22: CBPF to HCPN tranformation using ATL: mapping CBPB objects into HCPN elements

source element of the mapping: in our case, it is the *Composable business process fragment* meta-class belonging to the *CBPF metamodel*. Then we define the target element of the mapping: the *Petri Net* meta-class belonging to the *HCPN metamodel*. It is at this level where we also need to define which are the values of the newly created element, by mapping them to their corresponding attribute from the source model. In our example, the *name* attribute of the *Petri Net* element is the same with the *title* attribute of the *Composable business process fragment* element.

Another transformation example is presented in Figure 6.22, where an ATL rule is presented that transforms *CBPF objects* into *PetriNet elements*. The mapping rule is similar to the previous one.

CPN Tools:

CPN Tools [JKW07] provides an environment for editing and simulating HCPN models, and for verifying their correctness using state space analysis methods. CPN Tools combines powerful functionalities with a flexible user interface, containing improved interaction techniques, as well as different types of graphical feedback which keep the user informed of the status of syntax checks, simulations, etc.

CPN Tools basically consists of two components: a graphical editor and a simulator daemon. The editor allows users to interactively construct a CPN model that is transmitted to the simulator, which checks it for syntactical errors and generates model-specific code to simulate the CPN model. The editor invokes the generated simulator code and presents results graphically. The editor can load and save models using an XML format. A snapshot of the CPN Tools interface is presented in Figure 6.23.

One of the main features of CPN Tools is to allow users to create and edit HCPNs in an easy, fast and flexible manner. While a net is being edited, CPN Tools assists the user in a number of different ways, e.g. by providing a variety of graphical feedback regarding the

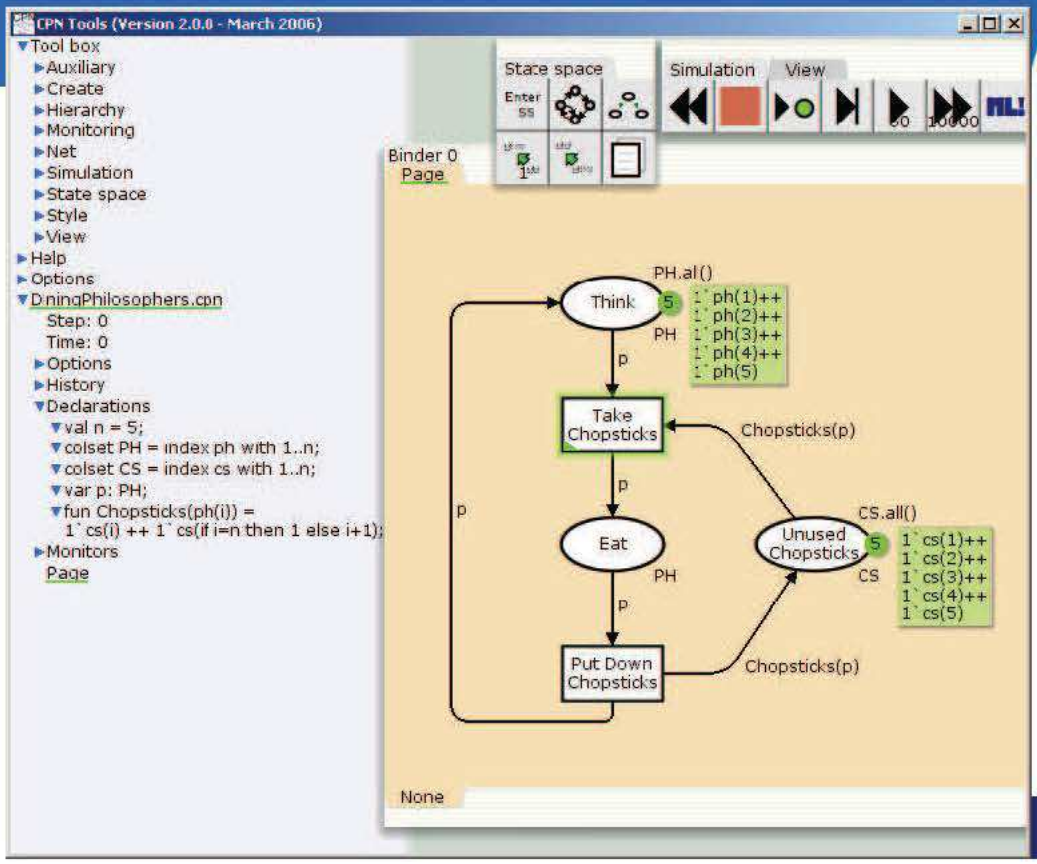


Fig. 6.23: Screen-shot of CPN Tools interface

syntax of the net and the status of the tool, or by automatically aligning objects in some situations. The syntax of a net is checked and simulation code for the net is automatically generated while the net is being constructed. *Create tools* are used to create HCPNt elements. All net elements can be created using palettes, tool-glasses and marking menus. *Style tools* can be used to change the style of any net element. *View tools* are used to define groups and to zoom in and out on a page. *Hierarchy tools* are used to create hierarchical CP-nets.

Another important feature of CPN Tools is its support for *syntax check and code generation.* The users invoke syntax checks explicitly, either through a command in a menu or through a switch to the simulation mode. In response to requests from users, this explicit syntax check has been eliminated, and CPN Tools instead features a syntax check that automatically runs in the background. *Code generation* is connected to the syntax check. When portions of a net are found to be syntactically correct, the necessary simulation code is automatically generated incrementally.

Simulations are controlled using the *Simulation tools.* The basic functionalities offered are: *rewind* (returns the net to its initial marking), *single-step tool* (causes one enabled transition to occur), *play tool* (will execute a user-defined number of steps). *Simulation feedback* is updated during the syntax check and during simulations. Green circles indicate how many tokens are currently on each place, and current markings appear in green text boxes next to the places. Green halos are used to indicate enabled transitions.

CPN Tools also contains facilities for *generating and analysing full and partial state spaces* for CP-nets. The provided state space tools are: *EnterStateSpace tool* (is used first to generate net-specific code necessary for generating a state space), *CalcSS tool* (generates the state space), *CalcSCC tool* (calculates the strongly connected component graph of the state space). Two tools exist for switching between the simulator and a state space. Standard state space reports can be generated automatically and saved using the *SaveReport tool.* The first step when conducting state space analysis is usually to ask for a *state space report*, which provides some basic information about the size of the state space and standard behavioural properties of the CPN model. The state space report provides the following information:

- *State space statistics* telling how large the state space is: number of nodes and arcs. We also get statistics about the SCC-graph;
- *Information about the boundedness properties:* best upper integer bound, best lower integer bounds, best upper multi-set bound and best lower multi-set bound;
- *Home properties* tell us about the existence of home markings;
- *Liveness properties* contain information about dead markings, dead and live transitions;

The aim of generating a state space is to check whether the considered model has certain properties. Some *standard queries* are relevant for many models, so CPN Tools supports that the results of the standard queries are automatically saved in a textual report.The user may also want to investigate properties that are not general enough to be part of the state space report. For this purpose a number of predefined *query functions* are available in CPN Tools that make it possible to write user-defined and model-dependent queries. CPN

Tools provides a general query language called the *CPN ML programming language*, based on the Standard ML language, for writing such generic or model specific queries. Query functions are typically used in auxiliary boxes, alone or as part of a larger ML expression. The box is evaluated by means of the ML Evaluate command. CPN Tools additionally contains a library that makes it possible to formulate queries in a temporal logic.

Using CPN Tools, several behavioural properties, presented in Section 5.3, can be verified. We present in the following how this is done using CPN Tools:

- *Reachability of end events:* we simply need to apply either the *Reachable'* or the *SccReachable'* functions, using the id of the start place as first parameter (usually 1) and the id of the last node as second parameter:

  $Reachable'(id - node_{start}, id - node_{end})$

- *Proper completion of a business process fragment:* in order to verify our property, we can simply use a combination of the home and dead marking query functions described above. For example, the simplest solution would be to first apply the *HomeMarking* function with the id of the last node as parameter, then apply the *DeadMarking* function with the same id as parameter:
  $HomeMarking(id - node_{end}), DeadMarking(id - node_{end})$

  However, if any of the two queries performed returns false, then the property is broken. They need to both be true in order for the property to be valid. Moreover, if the *ListDeadMarkings* function is used and it returns a list that contains more that the id of the end event, then a problem has been detected, as the marking of the end place needs to be the only dead marking in the net.

- *Reachability of composition interfaces:* the verification is actually done in two steps: first, based on the mapping, we determine the id for which we need to check the reachability (the place that follows the tagged transition as described above). Then we can apply the *Reachable* or *Reachable'* query functions with the id of the start node as parameter and the id determined in the first step as second parameter:
  $Reachable(id - node_{start}, id - node_{serched})$

  If the function returns true as a result, then we know that the tagged flow object for which we are performing the verification can be reached from the start event. However, for the property to be valid, it is required that this verification is true for all the flow objects that are part of the composition interface of a business process fragment.

- *Absence of dead tasks:* to verify that a business process fragment has no dead tasks, we can simply apply the *ListDeadTIs* function of the corresponding HCPN and check that the result returned by this query function is an empty list. In case the list returned as a result is non-empty, we know that the property is not satisfied.

- *Deadlock-free business process fragment:* in order to verify that a business process fragment is deadlock-free, we can simply apply the *ListDeadMarkings* query function on the corresponding HCPN. In order for the property to be fulfilled, the function has to return a list that only contains the last place of the net. In case the result returned is an empty list, then the property is satisfied but the *Proper completion property* is broken. If the result contains more elements, then the deadlock-freedom property is broken and we have identified the places in the net where the deadlocks occur.

Scheduler:

During the application engineering phase of the methodology, there is the need to create a composition workflow that gives the order in which the model compositions, in our case aspect weavings, will be performed. Moreover, the composition workflow needs to be parsed (interpreted), as this information is required by the weaver component. This functionality is provided by the *scheduler* module.

The weaving of aspects can raise potential conflicts; to deal with this, and because the order in which aspects are applied to a model can greatly influence the final result of a weaving, aspects need to be applied in a specific order. To facilitate this, a *Scheduler* module is issued, to manage, in an ordered and non-conflicting manner, the execution of both the join point detection and the weaving process itself, instead of having the user call the weaving process, one aspect at a time. The manner in which the *Scheduler* does this is by controlling the input files that are fed to the join point detector and by retrieving the output of the *Weaver* for re-processing, until all aspects have been processed and a final result has been achieved.

This modules is based on the part of the CBPF language that offers support for product derivation specification, already presented in Section 4.2.4 in Figure 4.9. The scheduler is actually a small domain specific language designed specifically for creating composition workflows. The language allows to define the following components:

- *Fragment place-holders:* for the composition workflow, business process fragments are seen as black boxes, we are not interested in their internal representations;
- *Operators:* the goal of the composition workflow is to specify the exact order in which process fragments are composed. It is essential to to be able to represent the different types of business process composition operators that can be applied;
- *Connectors:* we need to be able to represent the sequencing/flow of elements in the composition workflow.

A small graphical editor provides a simple palette that has graphical representations for all the above-mentioned components.

Fragment to aspect adapter:

This module of the SPLIT tool suite aims to transform a business process fragment (that may contain composition interfaces), in conjunction with a composition operator, into an aspect that can be processed by the *Weaver* plugin. The *Fragment to aspect adapter* module was developed by the *Centre de Recherche Public â Gabriel Lippmann* in the context of the SPLIT project.

The *Fragment to aspect adapter* module is an Eclipse plugin intended to be used in SPLIT tools suite. It is dedicated to business process fragments but can also be used with standard BPMN models. We have seen previously that in SPLIT tools suite, the composition of business process fragments is performed by means of aspect weaving. Because the *Weaver* module deals only with aspects, it is necessary to have a module that transforms business process fragments into aspects. Thus, the precise purpose of the *Fragment to aspect adapter* is to transforms a list of annotated business process fragments (used in conjunction with a composition operator for each fragment) into a list of aspects that conform to the *Aspect Model* defined in the *Weaver*.

ATL language is a hybrid of declarative and imperative programming. An ATL transformation program is composed of declarative rules that define the mapping between source and target models. When mapping is too complex to be performed declaratively or to ease programming, imperative constructs are possible.

5 Module architecture

The architecture is based on the composition operators.

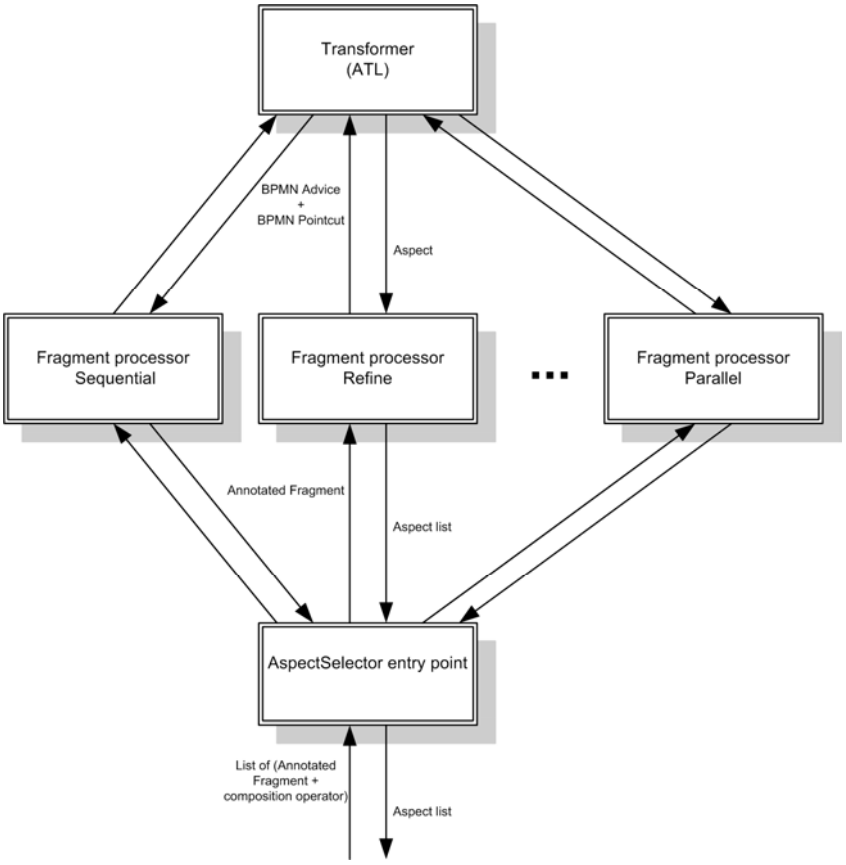


Figure 5 AspectSelector architecture

Fig. 6.24: Architecture of the *Fragment to aspect adapter* module

5.1 AspectSelector entry point

Input of the AspectSelector is a list of annotated fragment with their corresponding chosen composition operator.

The input and output data of this module are models that are conforms to their respective metamodels. The metamodels used in the *Fragment to aspect adapter* are conforms to Ecore (package org.eclipse.emf.ecore). The output format of the modules is an *aspect* composed of an *advice* and a *pointcut*. The main task of the *Fragment to aspect adapter* is to generate a *business process fragment advice* and a *business process fragment pointcut*, then link them together through a morphism and finally run a CBPF to AspectModel transformation using the Atlas Transformation Language (ATL).

The mapping between flow objects of the advice and flow objects of the pointcut is a morphism denoted g-morphism. This information is needed by the weaver because it acts as weaving instructions between the fragment and the base model. *G-morphismâ* is represented by an identifier put on any flow object. The identifier is added in CBPF as an extended metadata. The extended metadata chosen is an *EAnnotation*, part of Ecore model (package org.eclipse.emf.ecore.EAnnotation). *EAnnotation* is a key/value pair. The *Fragment to aspect adapter* use the string "AspectSelectorGMorphismID" as key and the g-morphism identifier as value.

The module also proposes a mapping between CBPF and AspectModel, performed by the ATL engine thanks to a given ATL language file. For each elements generated in the Aspect, some properties (AspectModel Propertie) are added like a unique identifier.

The overall architecture of the *Fragment to aspect adapter* module is presented in Figure

selected detection policies defined in Section 2.4.

Upon completion of the join point detection, an interface model is created, with the listing of all join points detected; to be processed in the next stages.

The format of this output is fixed by the meta-model presented in Figure 14, where a match is created for each join point detection, and is then comprised by all the pairings between the base and pointcut models, that produce the match. Because they are references to external models it is not possible to view in the figure the references of each pair, to the base and aspect model elements.

Figure 14: Join Point meta-model

Fig. 6.25: Join-Point meta-model

As a test module, the specific manner in which the detection occurs and what tools are used can be changed, by replacing the module, as long as the new module provides an output compliant with the meta-model presented in Figure 14.

6.24. It is based on the composition operators proposed in for the CBPF language. The input of the *AspectSelector entry point* is a list of annotated fragments with their corresponding chosen composition operator. The *AspectSelector* selects the right fragment processor. A fragment processor exists for each composition operator. The *AspectSelector* then aggregates the outputs of the Join Point detector, presented in Section 4.1.1, is processed to execute each weaving of the advice in the base model with the name of the *Aspect-*

4.1.2 Model weaver

In the Weaver module, the output of the Join Point detector, presented in Section 4.1.1, is processed to execute each weaving of the advice in the base model.

The mapping which the weaving takes place is done according to the strategies stated in Section 2.4.

We can also see that there exist a set of *fragment processors*, where each one is dedicated to only one composition operator. It builds a business process fragment advice from the annotated fragment: the business process fragment advice is a copy of the annotated fragment from which some composition interfaces are removed and some business process fragment elements are removed and/or added. G-morphism identifiers are generated and put in the advice. After the built of advice, it builds a business process fragment pointcut. The last step is to call the transformer with generated advice and pointcut. Custom fragment processors (composition operators) can be added to the *Fragment to aspect adapter* module. It is possible by modifying the sources.

The output generated by the Weaver module, is a complete model, which can either be the final model or be subject of further weaving, to integrate new aspects that have not been woven yet.

14

For some compositions, the processor needs to generate multiple advices for one fragment. That's why the processor could return a list of aspects. But in some cases, it needs to generate multiple pointcuts for one advice: in this particular case, the weaver will have to deal with the first aspect, and, if a join-point is found, it has to discard other aspects. To reflect this situation, the *AspectSelector* doesn't return a list of aspects but a list of *aspect choices*, each one is the list in which the weaver has to deal with no more than one matching aspect in the model.

Model composer module:

This module deals with the composition of business process fragments for obtaining the end result of the proposed SPL methodology. It consists of two separate plugins:

- *Model weaver*;
- *Join point detector*.

In the SPLIT tool, the model composition is implemented in terms of *aspect model weaving*. For this approach we need to consider first a *base model*, which for us is a business process we want to enrich with some *aspect*. Such an aspect contains a *pointcut* which gives the information of where the aspect will connect to in the base model, and an *advice*, which gives the the information of what is the contribution of the aspect. To be able to weave the aspect models into the base model, we also need to specify a pattern of where this weaving can occur in the base models. This pattern is called the *pointcut*. Each sub-section of the base model that is isomorphic to the pointcut is denoted as a join point, and these are the points where the weaving occurs.

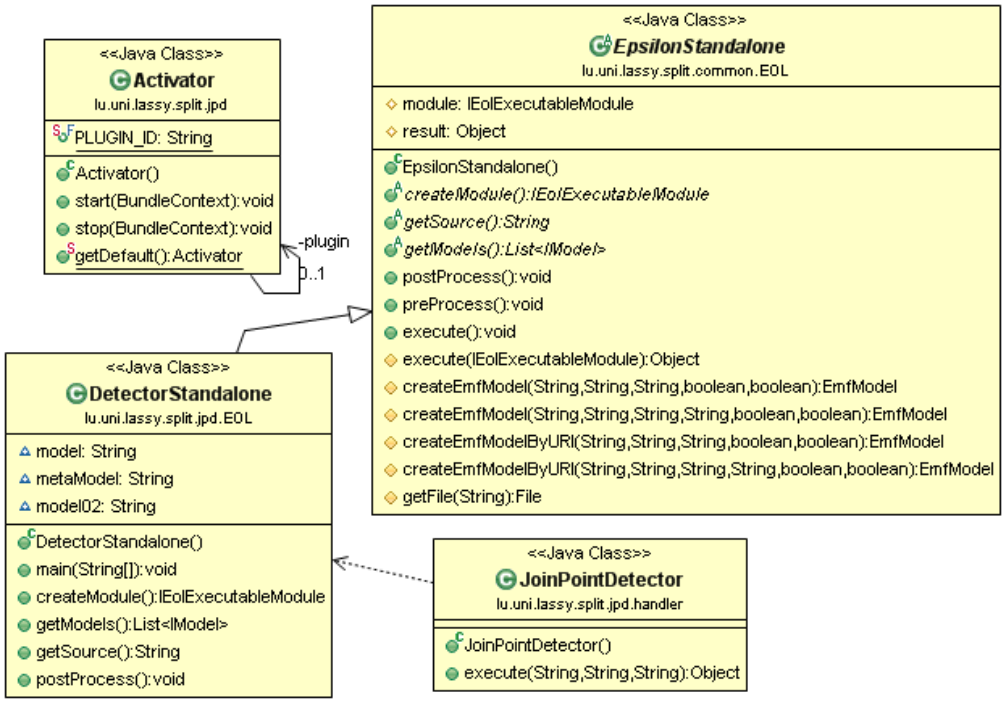


Fig. 6.26: Class Diagram of the Join Point Detector Module

The *Join point detector* plugin is responsible for detecting the join points of a single aspect in the base model, according to different detection policies. Upon completion of the join point detection, an interface model is created, with the listing of all join points detected; to be processed in the next stages. The format of this output is fixed by the meta-model presented in Figure 6.25, where a match is created for each join point detection, and is then comprised by all the pairings between the base and pointcut models, that produce the match. Because they are references to external models it is not possible to view in the figure the references of each pair, to the base and aspect model elements.

The class diagram for the main Join Point detection Plug-in is presented in Figure 6.26. In it, we can see the *DetectorStandalone* class that invokes the EOL interpreter, which inherits from *EpsilonStandalone*, a class based on the launching EOL from Java. The *JoinPointDetector* class serves as the interface of the module, providing the operations that should remain in alternative implementations of the Join Point Detection Module.

In the *Weaver* plugin, the output of the *Join Point detector* is processed to execute each weaving of the advice in the base model. The manner in which the weaving takes place is done according to the strategies stated in [MKKJ10]. The output generated by the weaver is a complete model, which can either be the final model or be subject of further weaving, to integrate new aspects that have not been woven yet. The weaver proposed in the SPLIT tool suite is a *generic model weaver*. This means that any type of model can be used as input for the weaver. However, in the context of this thesis, it is used with business process fragment models.

The design of the *Weaver* plugin is much like the one for the *Join Point Detector*, as it can be seen in Figure 6.27. It is composed of a class that invokes the EOL interpreter, in this case the class *WeaverStandalone*, who also inherits from *EpsilonStandalone*, and an

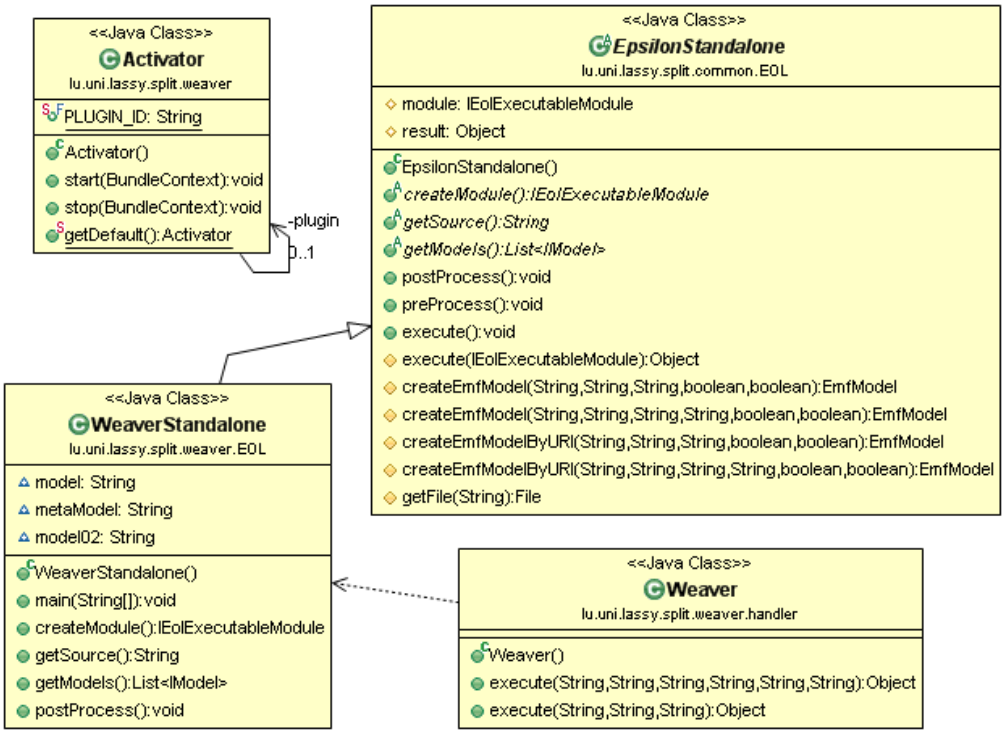
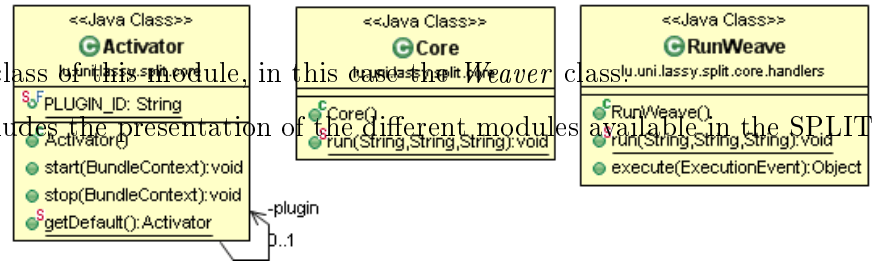


Fig. 6.27: Class Diagram of the Weaver Module



interface class of this module, in this case the *Weaver* class.

This concludes the presentation of the different modules available in the SPLIT tool suite.

7. PERSPECTIVES

Abstract

This Chapter describes possible improvements and extensions to the contributions of this thesis. Some of these extensions may be applied in the short term, whereas others may imply to explore new research areas. This chapter is divided in three parts: the first one focuses on extending the CBPF language with a set of composition operators; the second one addresses the use of aspect oriented techniques and in particular aspect weaving for composing business processes; the last part discusses the modelling of data fir business process fragments and its implications.

7.1 Defining composition operators for the CBPF language

One of the main contributions of this thesis is the CBPF language for modelling and composing business process fragments, presented in Chapter 4. CBPF was designed as the language support for the SPL methodology that we propose and presented in Chapter 3. It is created specifically for modelling composable business process fragments. A model driven approach is then followed for creating and specifying the CBPF domain specific language: abstract syntax, graphical concrete syntax and semantics definition.

The abstract syntax of the CBPF language is defined using a meta-model that specifies the components of the language and their relations in a concise manner. An important part of this meta-model is dedicated to presenting the support and concepts that CBPF offers for composing business process fragments. Thus, we introduced a set of binary composition operators: they take two business process fragments as input and produce a single process fragment as output of the composition. However, the composition operators that we proposed are only described at a hight level of abstraction. We present in general what each operator should do, define the necessary requirements that must be fulfilled for applying it, provide a textual notation for it and explain how the composition interfaces of the result are obtained.

We propose to extend the CBPF language specification and formally define the composition operators that we initially proposed. This allows to specify in a formal manner the syntax of each composition operator. It will also facilitate the exact understanding of how each operators behaves and which are the exact operations performed when applying that specific operator. Adding such a specification of the composition operators will enable the user to have a business process view of the composition. This means that the user will now be able to take two business process fragments as input, apply to them a particular composition operator and obtain a concrete business process fragment as a result.

This extension can be either be used individually, for composing different business process fragments using the proposed composition operators, or it may be used together with the

SPL methodology that we proposed in this thesis. During the last step of the methodology presented in Chapter 3, called the *product derivation specification*, we created a composition workflow which described which business process fragments need to be composed, in which order and using which specific composition operators. Thus, this composition workflow is also a high level and concise representation of the business process that we want to derive using the methodology. If the language extension is applied and a formal specification of the composition operators is available, we can also extend the methodology in the following manner: starting from the composition workflow, we parse it and apply all the composition operators in the order indicated in the workflow. The final result of this sequence of compositions is a new business process fragment which defines the behaviour of the product that we are deriving. Thus, it is now possible to have a business process view of the product that we are deriving. It is up to the user of the methodology is he wants to stop the derivation process at the level of the workflow as final product specification, or ha wants a more detailed and business process oriented view, in which case he can practically apply the operators.

Moreover, the specification of the composition operators at the level of business process fragments opens another interesting and challenging research perspective. In the presentation of the CBPF language from Chapter 4, the semantics of the composition operators was formally defined using a translational approach in terms of equivalent Petri net composition operators. All of the CBPF composition operators were mapped onto equivalent HCPN composition operators, which have a clear and well understood semantics. However, in this section we propose a new formalization of the CBPF language and of its composition operators using a set-based specification. The research question that implicitly arises is how can one prove that these two specifications are equivalent and lead to the same result. We thus need to prove that these two definitions of the composition operators lead to the same results when applied to the same business process fragments.

Using the approach presented in Chapter 4, given two business process fragments ($CBPF_1$) and ($CBPF_2$), in order to compose them using one of the proposed composition operators, we first need to apply the model-to-model transformation that was also proposed and obtain equivalent HCPN models. Then, the corresponding Petri net composition operator is applied for composing these HCPN models. Using the approach proposed in this section, we can directly apply the CBPF composition operator onto the ($CBPF_1$) and ($CBPF_2$) business process fragments. However, the challenge is to prove the equivalence of these two results. This needs to be done in a general case, for any two input business process fragments and for all of the proposed composition operators. This is a challenging and difficult task that we propose to address in the future.

In order to present the formal specification of the CBPF composition operators, we first need to choose an appropriate formalism in which the specification will be created. In our case, we propose to use a set-based mathematical specification. In a first step, we therefore propose a set-based specification of the CBPF abstract syntax. As the abstract syntax of the CBPF language was initially defined using a meta-model, it is quite straight-forward to obtain a set-based specification from it. Once this is done, we can start to formally define the set of composition operators that were proposed using the same set-based formalism. These two steps are presented in detail in the following sub-sections:

7.1.1 Mathematical specification of business process fragments

We want to provide a formal specification of the composition operators, in order to avoid any ambiguities in their definition. That is why we start by introducing a set-based formalization of the abstract syntax of business process fragments. This formalization will then be used for defining the functioning of the proposed composition operators.

Definition: We define the following notations for business process fragments:

- *Let $\mathcal{O}$ be the set of all objects that appear in all business process fragment diagrams*
- *Let $\mathcal{F}$ be the set of **flow objects** for all business process fragment diagrams: $\mathcal{F} \subseteq \mathcal{O}$*
- *Let $\mathcal{A}$ be the set of **activities** for all business process fragment diagrams: $\mathcal{A} \subseteq \mathcal{F}$*
- *Let $\mathcal{E}$ be the set of **events** for all business process fragment diagrams: $\mathcal{E} \subseteq \mathcal{F}$*
- *Let $\mathcal{G}$ be the set of **gateways** for all business process fragment diagrams: $\mathcal{G} \subseteq \mathcal{F}$*
- *The set of flow objects is partitioned into disjoint sets of activities $\mathcal{A}$, events $\mathcal{E}$, and gateways $\mathcal{G}$: $\mathcal{F} = \mathcal{A} \cup \mathcal{E} \cup \mathcal{G}$*
- *The set of events is partitioned into disjoint sets of start $\mathcal{E}_s$, intermediate $\mathcal{E}_i$, and end events $\mathcal{E}_e$: $\mathcal{E} = \mathcal{E}_s \cup \mathcal{E}_i \cup \mathcal{E}_e$*
- *The set of gateways is partitioned into disjoint sets of parallel $\mathcal{G}_p$, exclusive $\mathcal{G}_x$, inclusive $\mathcal{G}_i$, and complex* gateways *$\mathcal{G}_c$: $\mathcal{G} = \mathcal{G}_p \cup \mathcal{G}_x \cup \mathcal{G}_i \cup \mathcal{G}_c$*
- *The set of activities is partitioned into disjoint sets of tasks $\mathcal{T}$ and sub-processes $\mathcal{SP}$: $\mathcal{A} = \mathcal{T} \cup \mathcal{SP}$*
- *Let $\mathcal{A}r$ be the set of artifacts for all business process fragment diagrams: $\mathcal{A}r \subseteq \mathcal{O}$*
- *The set of artifacts is partitioned into disjoint sets of data objects $\mathcal{DO}$ and composition tags $\mathcal{CT}$: $\mathcal{A}r = \mathcal{DO} \cup \mathcal{CT}$*
- *Let $\mathcal{S}$ be the set of swimlanes for all business process fragment diagrams: $\mathcal{S} \subseteq \mathcal{O}$*

Using these notions, we propose the following definition for a business process fragment:

Definition: A business process fragment is a tuple BP (F, S, Ar, SF, MF, AS) where:

- *F is the set of **flow objects** of the business process fragment process, with $F \subseteq \mathcal{F}$*
- *S is the set of **swimlanes** of the business process fragment process, with $S \subseteq \mathcal{S}$*
- *Ar is the set of **artifacts** of the business process fragment process, with $Ar \subseteq \mathcal{A}r$*
- *$SF \subseteq F \times F$ defines a **sequence flow** relation between flow objects*
- *$MF \subseteq E \cup A \times E \cup A$ defines a **message flow** relation between events or activities*
- *$AS \subseteq F \times Ar$ defines an **association** relation between flow objects and artifacts*

For a business process fragment as defined above, we also have the following relations:

- The set of *flow objects* F is partitioned into disjoint sets of **activities** A, **events** E, and **gateways** G:
  $F = A \cup E \cup G$, where $A \subseteq \mathcal{A}, E \subseteq \mathcal{E}, G \subseteq \mathcal{G}$;
- The set of *artifacts* Ar is partitioned into disjoint set of **data objects** DO and **composition tags** CT:
  $Ar = DO \cup CT$, where $DO \subseteq \mathcal{D}O, CT \subseteq \mathcal{C}T$
- The et of *composition tags* CT is partitioned into disjoint sets of **input** CT_i and **output** CT_i composition tags: $CT = CT_i \cup CT_o$

Moreover, we also define *predecessor* and *successor* functions for flow objects, which will also be used later for the specification of the composition operators. The predecessor function returns the flow object connected by an input sequence flow relation with the flow object on which the function is applied. Similarly, the successor function returns the flow object connected by an output sequence flow relation with the flow object on which the function is applied.

Definition: For a business process fragment BP we define the following functions:

- Let $pred : F \rightarrow F, pred(x) = \{y|(y, x) \in SF\}$
- Let $succ : F \rightarrow F, succ(x) = \{y|(x, y) \in SF\}$

We also define a *tagging function* that returns the composition tag of a flow object, if it has one

Definition: For a business process fragment BP, we define the tagging function as:

- $Tag : F \rightarrow CT$, where $F \subseteq \mathcal{F}$

We can now formally define the concept of *composition interface* of a business process fragments:

Definition: The composition interface of a business process fragment is $I = I_i \cup I_o$, where:

- I_i *is the input composition interface:* $I_i = \{x|x \in F, Tag(x) \in CT_i\}$
- I_o *is the output composition interface:* $I_o = \{x|x \in F, Tag(x) \in CT_o\}$

Finally, we can now formally define the notion of *composable business process fragment*, used as input for our composition operators:

Definition: A composable business process fragment is a tuple
$CBPF = (F, S, Ar, SF, MF, AS, I, Tag)$, where (F, S, Ar, SF, MF, AS) defines a business process fragment, Tag is a tagging function returning the composition tags of flow objects, and I is the composition interface of the fragment.

For a composable business process fragment, we define the following auxiliary functions:

- $out : CBPF \rightarrow E_e, out(BP) = \{e|\, Tag(e) \in CT_{in}\}, E_e = E \cap \mathcal{E}_e$ returns the end events of a process tagged with an output composition interface.
- $in : CBPF \rightarrow E_e, out(BP) = \{e|\, Tag(e) \in CT_{out}\}, E_s = E \cap \mathcal{E}_s$ returns the start events of a process tagged with an input composition interface.

7.1.2 Proposed composition operators

All the composition operators we propose are *binary composition operators:* they take two business process fragments as input and produce a single process fragment as output of the composition.

For all the composition operators that will be presented, we use the following notations:

- Let $CBPF_1 = (F_1, S_1, Ar_1, SF_1, MF_1, AS_1, I_1)$ and $CBPF_2 = (F_2, S_2, Ar_2, SF_2, MF_2, AS_2, I_2)$ denote two composable business process fragments used as input for the composition operators.
- Let $CBPF_{res} = (F_{res}, S_{res}, Ar_{res}, SF_{res}, MF_{res}, AS_{res}, I_{res})$ denote the result of applying a composition operator.

Sequential composition operator:

Definition: Let $CBPF_1 = (F_1, S_1, Ar_1, SF_1, MF_1, AS_1, I_1)$ and $CBPF_2 = (F_2, S_2, Ar_2, SF_2, MF_2, AS_2, I_2)$ be two business process fragments. The result of applying the sequential composition operator on the process fragments $(CBPF_1)$ and $(CBPF_2)$, denoted $seq(CBPF_1, CBPF_2)$, is a new business process fragment $CBPF_{res} = (F_{res}, S_{res}, Ar_{res}, SF_{res}, MF_{res}, AS_{res}, I_{res})$ where:

- The flow objects of the result contain the union of the flow objects from the input models, from which we remove the end event of $CBPF_1$ tagged with a composition interface, and the start event of $CBPF_2$:
  $F_{res} = F_1 \cup F_2 \setminus \{out(CBPF_1), in(CBPF_2)\}$
- For the resulting sequence flow, we need to disconnect and from the initial models, then connect together the remaining process fragments:
  $SF_{res} = SF_1 \cup SF_2 \setminus \{(pred(out(CBPF_1)), out(CBPF_1)), (in(CBPF_2), succ(in(CBPF_2)))\} \cup (pred(out(CBPF_1)), succ(in(CBPF_2)))$
- The swimlanes, artifacts, message flow and associations of the result are the union of their counterparts from the input processes:
  $S_{res} = S_1 \cup S_2$
  $Ar_{res} = Ar_1 \cup Ar_2$
  $MF_{res} = MF_1 \cup MF_2$
  $AS_{res} = AS_1 \cup AS_2$
- The composition interface of the result is the union of the interfaces of the input models, from which we need to remove and :
  $I_{res} = I_{i-1} \setminus \{out(CBPF_1)\} \cup I_{i-2} \setminus \{in(CBPF_2)\}$

The general functioning of the operator is graphically depicted in Figure 7.1.

Parallel composition operator:

Definition: Let $CBPF_1 = (F_1, S_1, Ar_1, SF_1, MF_1, AS_1, I_1)$ and $CBPF_2 = (F_2, S_2, Ar_2, SF_2, MF_2, AS_2, I_2)$ be two business process fragments. The result

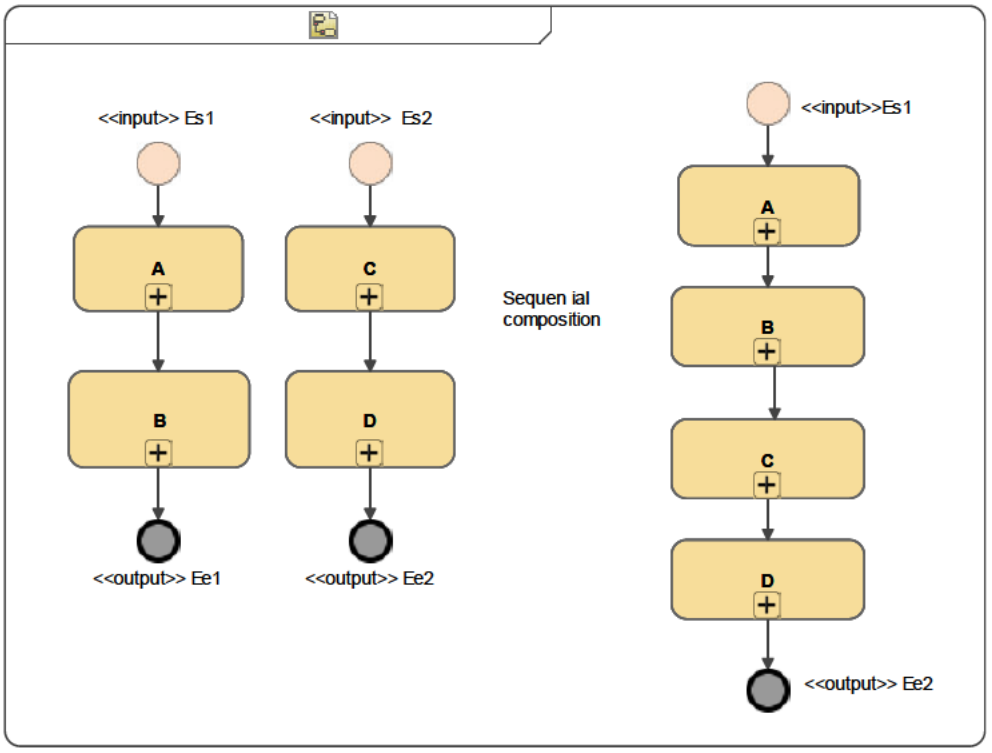


Fig. 7.1: Sequential composition operator for business process fragments

of applying the parallel composition operator on the process fragments $(CBPF_1)$ and $(CBPF_1)$, denoted $par(CBPF_1, CBPF_2)$, is a new business process fragment $CBPF_{res} = (F_{res}, S_{res}, Ar_{res}, SF_{res}, MF_{res}, AS_{res}, I_{res})$ where:

- The result contains the union of all activities from the input process fragments: $A_{res} = A_1 \cup A_2$
- The result contains the union of the events of the two input process fragments, from which we need to remove the start events of $CBPF_1$ and $CBPF_2$ and their end events tagged with composition interfaces, then add a new start and end event: $E_{res} = E_1 \cup E_2 \backslash \{in(CBPF_1), out(CBPF_1), in(CBPF_2), out(CBPF_2)\} \cup \{start_{new}, end_{new}\}$, where $E_1 \subseteq F_1, E_2 \subseteq F_2, start_{new} \in \mathcal{E}_s, end_{new} \in \mathcal{E}_e$
- To obtain the gateways of the result, we take the union of the gateways of the input process fragments and add two new gateways, a splitting parallel one and a merging parallel one: $G_{res} = G_1 \cup G_2 \cup \{g_1, g_2\}$, where $G_1 \subseteq F_1, G_2 \subseteq F_2$ and $g_1, g_2 \in \mathcal{G}_p$
- The sequence flow of the result is obtained from the union of sequence flows of the input fragments, from which we first need to disconnect the start and end events, then connect the new start and end events to the newly introduced gateways, then finally connect these gateways to the remaining parts of the input process fragments: For simplicity and to improve the understanding, we use the following notations: $in_1 = in(CBPF_1), in_2 = in(CBPF_2), out_1 = out(CBPF_1), out_2 = out(CBPF_2)$ $SF_{res} = SF_1 \cup SF_2 \backslash \{(in_1, succ(in_1)), (pred(out_1), out_1), (in_2, succ(in_2)), (pred(out_2), out_2)\} \cup \{(start_{new}, g_1), (g_2, end_{new}), (g_1, succ(in_1)), (g_1, succ(in_2)), (pred(out_2), g_2), (pred(out_1), g_2)\}$
- The swimlanes, artifacts, message flow and associations of the result are the union of their counterparts from the input processes:

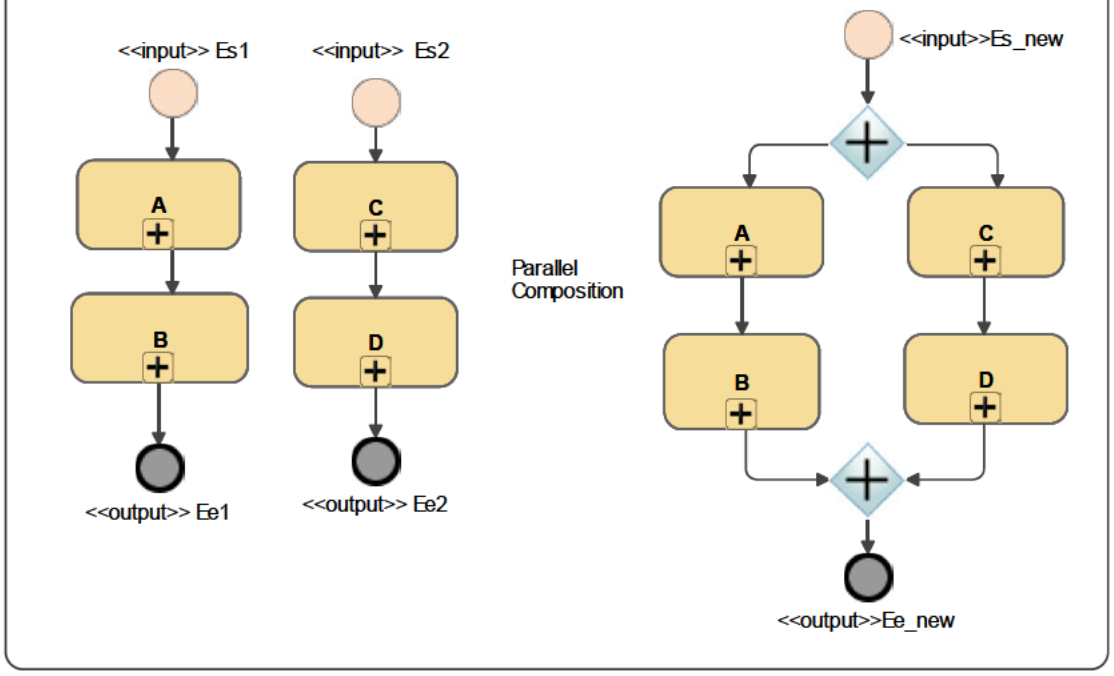


Fig. 7.2: Parallel composition operator for business process fragments

$S_{res} = S_1 \cup S_2$
$Ar_{res} = Ar_1 \cup Ar_2$
$MF_{res} = MF_1 \cup MF_2$
$AS_{res} = AS_1 \cup AS_2$

- The composition interface of the result contains the union of interfaces of the input models, from which we remove the start events and end events tagged with composition interfaces of $CBPF_1$ and $CBPF_2$, and add an output composition tag at the newly introduced end event and an input composition tag at the newly introduced start event:

  $I_{res} = I_1 \cup I_1 \setminus \{in(CBPF_1), out(CBPF_1), in(CBPF_2), out(CBPF_2)\} \cup \{start_{new}, end_{new}\}$
  where $start_{new} \in \mathcal{E}_s, end_{new} \in \mathcal{E}_e$

The general functioning of the operator is graphically depicted in Figure 7.1.

Exclusive choice composition operator:

Definition: Let $CBPF_1 = (F_1, S_1, Ar_1, SF_1, MF_1, AS_1, I_1)$ and $CBPF_2 = (F_2, S_2, Ar_2, SF_2, MF_2, AS_2, I_2)$ be two business process fragments. The result of applying the exclusive choice composition operator on the process fragments $(CBPF_1)$ and $(CBPF_1)$, denoted $excl(CBPF_1, CBPF_2)$, is a new business process fragment $CBPF_{res} = (F_{res}, S_{res}, Ar_{res}, SF_{res}, MF_{res}, AS_{res}, I_{res})$ where:

- The result contains the union of all activities from the input process fragments: $A_{res} = A_1 \cup A_2$
- The result contains the union of the events of the two input process fragments, from which we need to remove the start events of $CBPF_1$ and $CBPF_2$ and their end events

tagged with composition interfaces, then add a new start and end event:
$E_{res} = E_1 \cup E_2 \setminus \{in(CBPF_1), out(CBPF_1), in(CBPF_2), out(CBPF_2)\} \cup \{start_{new}, end_{new}\}$,
where $E_1 \subseteq F_1, E_2 \subseteq F_2, start_{new} \in \mathcal{E}_s, end_{new} \in \mathcal{E}_e$

- To obtain the gateways of the result, we take the union of the gateways of the input process fragments and add two new gateways, a splitting exclusive one and a merging exclusive one:
  $G_{res} = G_1 \cup G_2 \cup \{g_1, g_2\}$, where $G_1 \subseteq F_1, G_2 \subseteq F_2$ and $g_1, g_2 \in \mathcal{G}_x$
- The sequence flow of the result is obtained from the union of sequence flows of the input fragments, from which we first need to disconnect the start and end events, then connect the new start and end events to the newly introduced gateways, then finally connect these gateways to the remaining parts of the input process fragments:
  For simplicity and to improve the understanding, we use the following notations:
  $in_1 = in(CBPF_1), in_2 = in(CBPF_2), out_1 = out(CBPF_1), out_2 = out(CBPF_2)$
  $SF_{res} = SF_1 \cup SF_2 \setminus \{(in_1, succ(in_1)), (pred(out_1), out_1), (in_2, succ(in_2)), (pred(out_2), out_2)\} \cup$
  $\{(start_{new}, g_1), (g_2, end_{new}), (g_1, succ(in_1)), (g_1, succ(in_2)), (pred(out_2), g_2), (pred(out_1), g_2)\}$
- The swimlanes, artifacts, message flow and associations of the result are the union of their counterparts from the input processes:

  $S_{res} = S_1 \cup S_2$
  $Ar_{res} = Ar_1 \cup Ar_2$
  $MF_{res} = MF_1 \cup MF_2$
  $AS_{res} = AS_1 \cup AS_2$
- The composition interface of the result contains the union of interfaces of the input models, from which we remove the start events and end events tagged with composition interfaces of $CBPF_1$ and $CBPF_2$, and add an output composition tag at the newly introduced end event and an input composition tag at the newly introduced start event:

  $I_{res} = I_1 \cup I_1 \setminus \{in(CBPF_1), out(CBPF_1), in(CBPF_2), out(CBPF_2)\} \cup \{start_{new}, end_{new}\}$
  where $start_{new} \in \mathcal{E}_s, end_{new} \in \mathcal{E}_e$

The general functioning of the operator is graphically depicted in Figure 7.3.

Choice composition operator:

Definition: Let $CBPF_1 = (F_1, S_1, Ar_1, SF_1, MF_1, AS_1, I_1)$ and
$CBPF_2 = (F_2, S_2, Ar_2, SF_2, MF_2, AS_2, I_2)$ be two business process fragments. The result of applying the choice composition operator on the process fragments $(CBPF_1)$ and $(CBPF_1)$, denoted $cho(CBPF_1, CBPF_2)$, is a new business process fragment
$CBPF_{res} = (F_{res}, S_{res}, Ar_{res}, SF_{res}, MF_{res}, AS_{res}, I_{res})$ where:

- The result contains the union of all activities from the input process fragments:
  $A_{res} = A_1 \cup A_2$
- The result contains the union of the events of the two input process fragments, from which we need to remove the start events of $CBPF_1$ and $CBPF_2$ and their end events tagged with composition interfaces, then add a new start and end event:
  $E_{res} = E_1 \cup E_2 \setminus \{in(CBPF_1), out(CBPF_1), in(CBPF_2), out(CBPF_2)\} \cup \{start_{new}, end_{new}\}$,
  where $E_1 \subseteq F_1, E_2 \subseteq F_2, start_{new} \in \mathcal{E}_s, end_{new} \in \mathcal{E}_e$

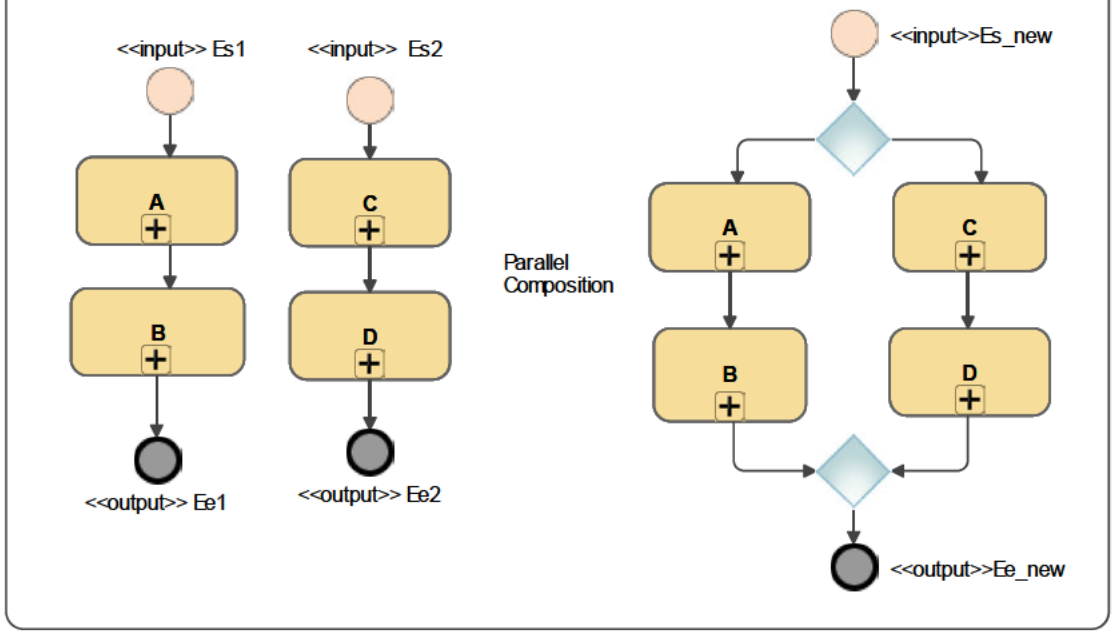


Fig. 7.3: Exclusive choice composition operator for business process fragments

- To obtain the gateways of the result, we take the union of the gateways of the input process fragments and add two new gateways, a splitting inclusive one and a merging inclusive one:
  $G_{res} = G_1 \cup G_2 \cup \{g_1, g_2\}$, where $G_1 \subseteq F_1, G_2 \subseteq F_2$ and $g_1, g_2 \in \mathcal{G}_i$

- The sequence flow of the result is obtained from the union of sequence flows of the input fragments, from which we first need to disconnect the start and end events, then connect the new start and end events to the newly introduced gateways, then finally connect these gateways to the remaining parts of the input process fragments:
  For simplicity and to improve the understanding, we use the following notations:
  $in_1 = in(CBPF_1), in_2 = in(CBPF_2), out_1 = out(CBPF_1), out_2 = out(CBPF_2)$
  $SF_{res} = SF_1 \cup SF_2 \setminus \{(in_1, succ(in_1)), (pred(out_1), out_1), (in_2, succ(in_2)), (pred(out_2), out_2)\} \cup$
  $\{(start_{new}, g_1), (g_2, end_{new}), (g_1, succ(in_1)), (g_1, succ(in_2)), (pred(out_2), g_2), (pred(out_1), g_2)\}$

- The swimlanes, artifacts, message flow and associations of the result are the union of their counterparts from the input processes:

  $S_{res} = S_1 \cup S_2$
  $Ar_{res} = Ar_1 \cup Ar_2$
  $MF_{res} = MF_1 \cup MF_2$
  $AS_{res} = AS_1 \cup AS_2$

- The composition interface of the result contains the union of interfaces of the input models, from which we remove the start events and end events tagged with composition interfaces of $CBPF_1$ and $CBPF_2$, and add an output composition tag at the newly introduced end event and an input composition tag at the newly introduced start event:

  $I_{res} = I_1 \cup I_1 \setminus \{in(CBPF_1), out(CBPF_1), in(CBPF_2), out(CBPF_2)\} \cup \{start_{new}, end_{new}\}$
  where $start_{new} \in \mathcal{E}_s, end_{new} \in \mathcal{E}_e$

The general functioning of the operator is graphically depicted in Figure 7.4.

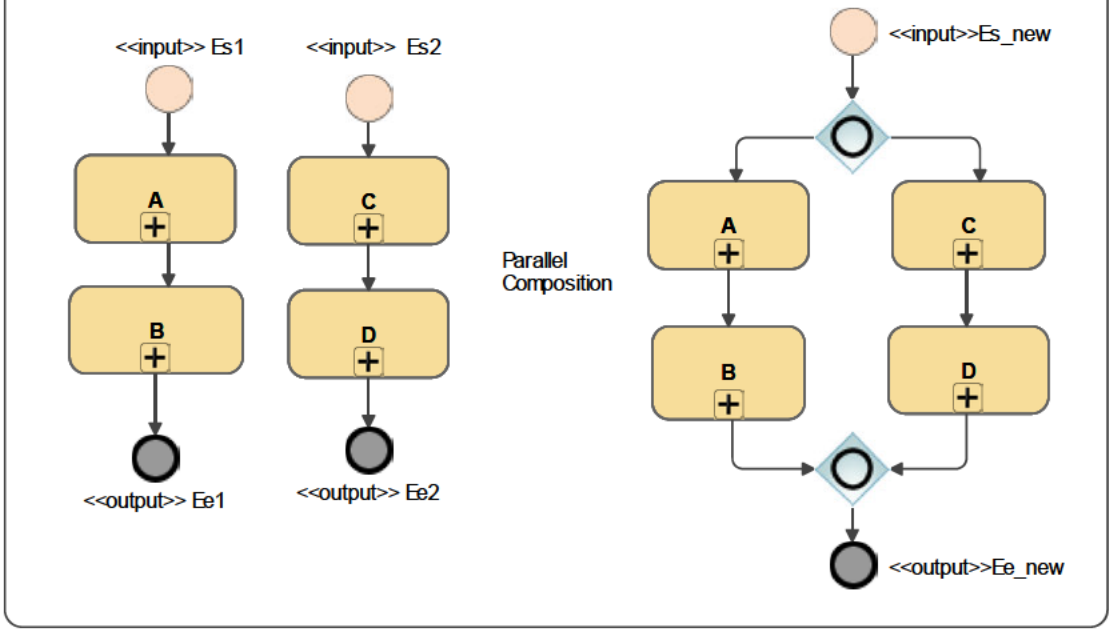


Fig. 7.4: Choice composition operator for business process fragments

Unordered (arbitrary) sequence composition operator:

Definition: Let $CBPF_1 = (F_1, S_1, Ar_1, SF_1, MF_1, AS_1, I_1)$ and $CBPF_2 = (F_2, S_2, Ar_2, SF_2, MF_2, AS_2, I_2)$ be two business process fragments. The result of applying the arbitrary sequence composition operator on the process fragments $(CBPF_1)$ and $(CBPF_1)$, denoted $arb(CBPF_1, CBPF_2)$, is a new business process fragment $CBPF_{res} = (F_{res}, S_{res}, Ar_{res}, SF_{res}, MF_{res}, AS_{res}, I_{res})$ where:

- We need first to make copies of the activities of $CBPF_1$ and $CBPF_2$. Then, the result will contain the union of all activities from the input process fragments, together with the previously created copies of the activities:
  $A_{res} = A_1 \cup A_2 \cup A'_1 \cup A'_2$, where A'_1, A'_2 are exact copies of A_1, A_2 respectively
- We fist make copies of the intermediate events of the two input process fragments. The result will then contain the union of the events of the two input process fragments, from which we need to remove the start events of $CBPF_1$ and $CBPF_2$ and their end events tagged with composition interfaces. We also add the previously created copies of the intermediate events. Finally, we add a new start and end event. :
  $E_{res} = E_1 \cup E_2 \setminus \{in(CBPF_1), out(CBPF_1), in(CBPF_2), out(CBPF_2)\} \cup \{start_{new}, end_{new}\} \cup Ei'_1 \cup Ei'_2$, where $E_1 \subseteq F_1, E_2 \subseteq F_2, start_{new} \in \mathcal{E}_s, end_{new} \in \mathcal{E}_e, Ei_1 = E_1 \cap \mathcal{E}_i, Ei_2 = E_2 \cap \mathcal{E}_i$ and Ei'_1, Ei'_2 are exact copies of Ei_1, Ei_2 respectively
- As before, we first make copies of the gateways of the two input process fragments. To obtain the gateways of the result, we take the union of the gateways of the input process fragments, add the previously created gateway copies, then add two new gateways, a splitting inclusive and a merging inclusive one:
  $G_{res} = G_1 \cup G_2 \cup G'_1 \cup G'_2 \cup \{g_1, g_2\}$, where $G_1 \subseteq F_1, G_2 \subseteq F_2$, $g_1, g_2 \in \mathcal{G}_i$ and G'_1, G'_2 are exact copies of G_1, G_2 respectively
- As before, we first make copies of the sequence flows of the two input fragments. Then, the sequence flow of the result is obtained from the union of sequence flows of

the input fragments, together with the previously created copies of sequence flows. We then need to remove the sequence flows connecting the start and end events, from both the original fragments and the copies. The next step is to add a new sequence flow relation connecting the end of the first fragment with the beginning of the second. Similarly, we add a new sequence flow connecting the end of the copy of the second fragment with the start of the copy of the first fragment. The fragments thus obtained need to be connected with the splitting and merging inclusive gateways. Finally, the last step is to connect using sequence flow the new start and end events to the gateways:
For simplicity and to improve the understanding, we use the following notations:
$in_1 = in(CBPF_1), in_2 = in(CBPF_2), out_1 = out(CBPF_1), out_2 = out(CBPF_2)$
$SF_{res} = SF_1 \cup SF_2 \cup SF_1' \cup SF_2' \setminus \{(in_1, succ(in_1)), (pred(out_1), out_1), (in_2, succ(in_2)),$
$(pred(out_2), out_2), (in_1', succ(in_1')), (pred(out_1'), out_1'), (in_2', succ(in_2')), (pred(out_2), out_2)\} \cup$

$\{(pred(out_1), succ(in_2'))\} \cup \{(pred(out_2'), succ(in_1'))\} \cup \{(g_1, succ(in_1)), (g_1, succ(in_2'))\} \cup$

$\{(pred(out_2), g_2), (pred(out_1'), g_2)\} \cup \{(start_{new}, g_1), (g_2, end_{new})\}$

- The swimlanes of the result are the union of their counterparts from the input processes:
$S_{res} = S_1 \cup S_2$

- We make first a copy of the artifacts from the two input processes. The artifacts of the result are the union of their counterparts from the input processes to which we add the previously created copies of artifacts:
$Ar_{res} = Ar_1 \cup Ar_2 \cup Ar_1' \cup Ar_2'$, where Ar_1', Ar_2' are exact copies of Ar_1, Ar_2 respectively

- We make first a copy of the message flows from the two input processes. The message flow of the result are the union of their counterparts from the input processes to which we add the previously created copies of message flows:
$MF_{res} = MF_1 \cup MF_2 \cup MF_1' \cup MF_2'$, where MF_1', MF_2' are exact copies of MF_1, MF_2 respectively

- We make first a copy of the associations from the two input processes. The associations of the result are the union of their counterparts from the input processes to which we add the previously created copies of associations:
$AS_{res} = AS_1 \cup AS_2 \cup AS_1 \cup AS_2$, where AS_1', AS_2' are exact copies of AS_1, AS_2 respectively

- As before, we first make a copy of the composition interfaces of the input fragments. The composition interface of the result contains the union of interfaces of the input models, together with the previously created composition interface copies, from which we remove the start events and end events tagged with composition interfaces of $CBPF_1, CBPF_2, CBPF_1',$
$CBPF_2'$, and add an output composition tag at the newly introduced end event and an input composition tag at the newly introduced start event:

$I_{res} = I_1 \cup I_2 \cup I_1' \cup I_2 \setminus \{in(CBPF_1), out(CBPF_1), in(CBPF_2), out(CBPF_2), in(CBPF_1'),$
$out(CBPF_1'), in(CBPF_2'), out(CBPF_2')\} \cup \{start_{new}, end_{new}\}$ where $start_{new} \in \mathcal{E}_s, end_{new} \in \mathcal{E}_e$

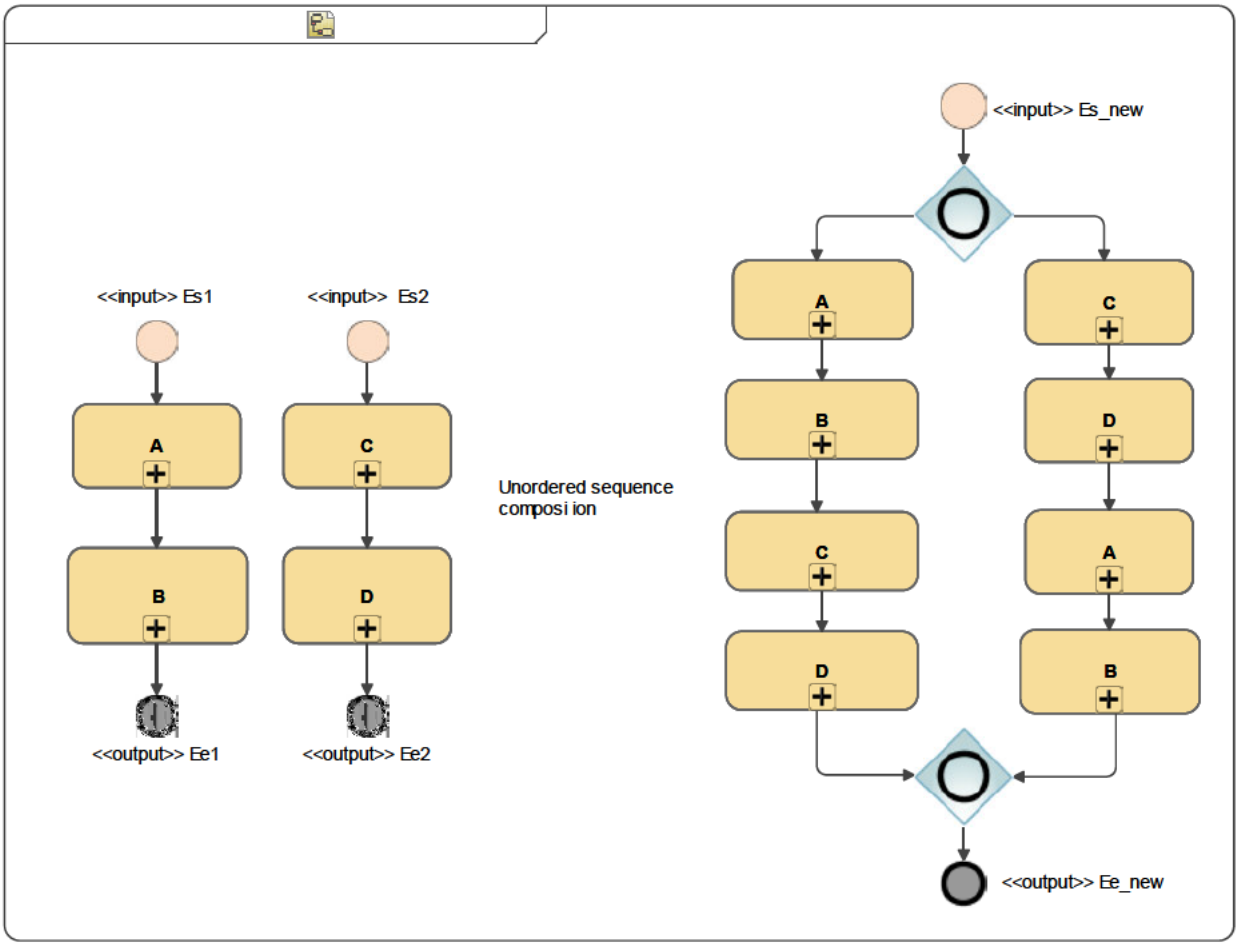


Fig. 7.5: Unordered (arbitrary) sequence composition operator for business process fragments

The general functioning of the operator is graphically depicted in Figure 7.5. At a closer analysis, this operator is a composed one, which can be replaced by using the sequential and choice operators together in the following manner: $seq(CBPF_1, CBPF_2)$ and $seq(CBPF_2, CBPF_1)$, followed by a choice composition of those results. Using the notation introduced, we obtain:

$$arb(CBPF_1, CBPF_2) = cho(seq(CBPF_1, CBPF_2), seq(CBPF_2, CBPF_1))$$

Parallel with communication composition operator:

Definition: Let $CBPF_1 = (F_1, S_1, Ar_1, SF_1, MF_1, AS_1, I_1)$ and $CBPF_2 = (F_2, S_2, Ar_2, SF_2, MF_2, AS_2, I_2)$ be two business process fragments. The result of applying the parallel with communication composition operator on the process fragments $(CBPF_1)$ and $(CBPF_1)$, denoted $parC(CBPF_1, CBPF_2)$, is a new business process fragment $CBPF_{res} = (F_{res}, S_{res}, Ar_{res}, SF_{res}, MF_{res}, AS_{res}, I_{res})$ where:

- The result contains the union of all activities from the input process fragments: $A_{res} = A_1 \cup A_2$
- The result contains the union of the events of the two input process fragments, from which we need to remove the start events of $CBPF_1$ and $CBPF_2$ and their end events tagged with composition interfaces, then add a new start and end event:

$E_{res} = E_1 \cup E_2 \backslash \{in(CBPF_1), out(CBPF_1), in(CBPF_2), out(CBPF_2)\} \cup \{start_{new}, end_{new}$,
where $E_1 \subseteq F_1, E_2 \subseteq F_2, start_{new} \in \mathcal{E}_s, end_{new} \in \mathcal{E}_e$

- To obtain the gateways of the result, we take the union of the gateways of the input business process fragments and add two new gateways, a splitting parallel one and a merging parallel one:
  $G_{res} = G_1 \cup G_2 \cup \{g_1, g_2\}$, where $G_1 \subseteq F_1, G_2 \subseteq F_2$ and $g_1, g_2 \in \mathcal{G}_p$

- The sequence flow of the result is obtained from the union of sequence flows of the input process fragments, from which we first need to disconnect the start and end events, then connect the new start and end events to the newly introduced gateways. Finally, we connect these gateways to the remaining parts of the input process fragments:
  For simplicity and to improve the understanding, we use the following notations:
  $in_1 = in(CBPF_1), in_2 = in(CBPF_2), out_1 = out(CBPF_1), out_2 = out(CBPF_2)$
  $SF_{res} = SF_1 \cup SF_2 \backslash \{(in_1, succ(in_1)), (pred(out_1), out_1), (in_2, succ(in_2)), (pred(out_2), out_2)\} \cup$
  $\{(start_{new}, g_1), (g_2, end_{new}), (g_1, succ(in_1)), (g_1, succ(in_2)), (pred(out_2), g_2), (pred(out_1), g_2)\}$

- The swimlanes, artifacts, and associations of the result are the union of their counterparts from the input processes:

  $S_{res} = S_1 \cup S_2$
  $Ar_{res} = Ar_1 \cup Ar_2$
  $AS_{res} = AS_1 \cup AS_2$

- The message flow of the result is the union if the message flows of the input process fragments to which we add a set of message exchanges between elements from the two fragments that belong to the same pair in SCE:

  $MF_{res} = MF_1 \cup MF_2 \cup \{(x, y) | (x, y) \in SCE, x \in I_o 1, y \in I_i 2\} \cup \{(x, y) | (x, y) \in SCE, x \in I_o 2, y \in I_i 1\}$

- The composition interface of the result contains the union of interfaces of the input models, from which we remove the start events and end events tagged with composition interfaces of $CBPF_1$ and $CBPF_2$, and add an output composition tag at the newly introduced end event and an input composition tag at the newly introduced start event:

  $I_{res} = I_1 \cup I_1 \backslash \{in(CBPF_1), out(CBPF_1), in(CBPF_2), out(CBPF_2)\} \cup \{start_{new}, end_{new}\}$
  where $start_{new} \in \mathcal{E}_s, end_{new} \in \mathcal{E}_e$

The general functioning of the operator is graphically depicted in Figure 7.6.

Refinement composition operator:

To facilitate the description of the operator, we use of the following notations: let *comp1* be the activity of $CBPF_1$ tagged with either and input or output composition interface.

Definition: Let $CBPF_1 = (F_1, S_1, Ar_1, SF_1, MF_1, AS_1, I_1)$ and
$CBPF_2 = (F_2, S_2, Ar_2, SF_2, MF_2, AS_2, I_2)$ be two business process fragments. The result of applying the refinement composition operator on the process fragments $(CBPF_1)$ and $(CBPF_1)$, denoted $ref(CBPF_1, CBPF_2)$, is a new business process fragment
$CBPF_{res} = (F_{res}, S_{res}, Ar_{res}, SF_{res}, MF_{res}, AS_{res}, I_{res})$ where:

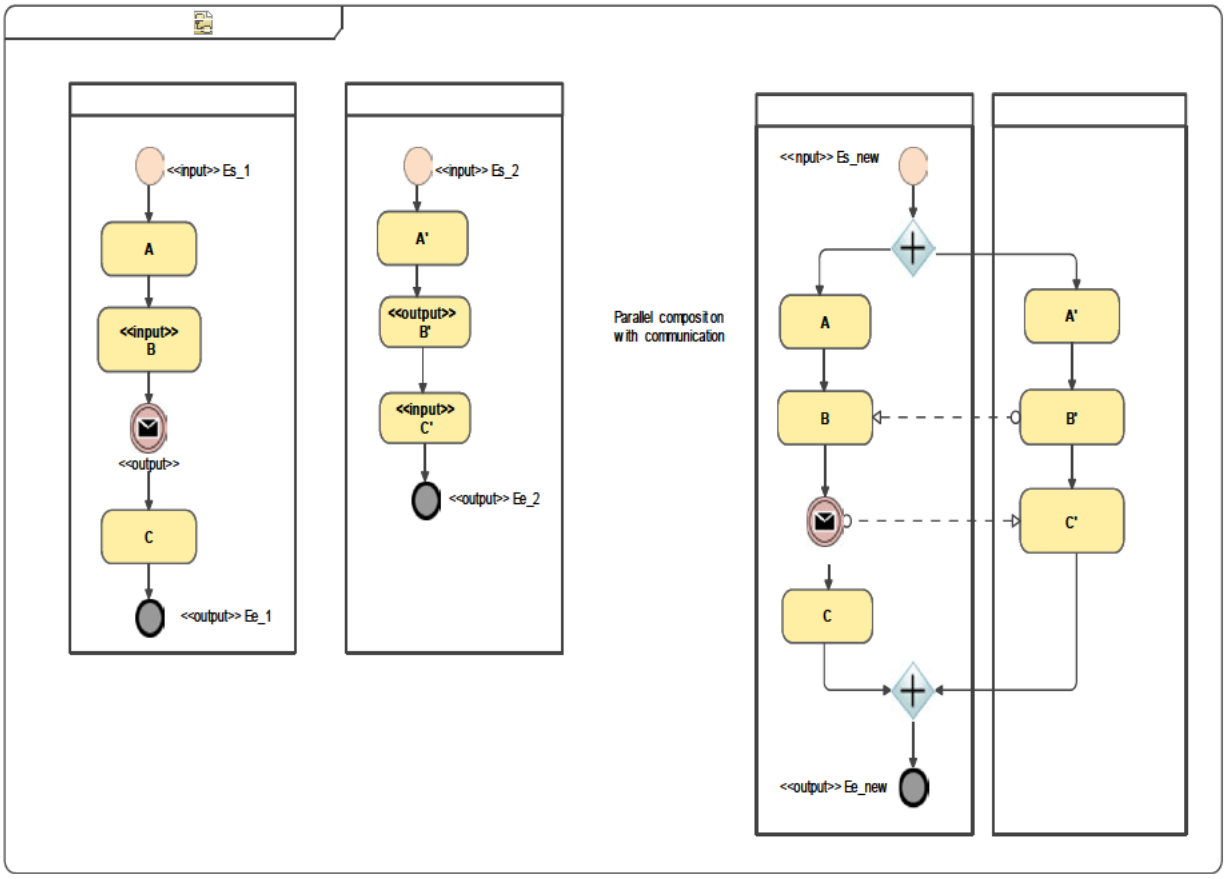


Fig. 7.6: Parallel with communication composition operator for business process fragments

- The flow objects of the result contain the union of the flow objects of the input process fragments, from which we have to remove the activity from $CBPF_1$ tagged with a composition interface, and also the start event and end event from BP_2 tagged with composition interfaces:

  $F_{res} = F_1 \cup F_2 \setminus \{comp1, in(CBPF_2), out(CBPF_2)\}$, where $comp1 \in F_1$

- The sequence flow of the result is obtained by first disconnecting the tagged activity from $CBPF_1$, then connecting the resulting upper process fragments to the first flow object of $CBPF_2$ and the resulting lower process fragment to the last flow object of $CBPF_2$:
  For simplicity and to improve the understanding, we use the following notations:
  $in_1 = in(CBPF_1), in_2 = in(CBPF_2), out_1 = out(CBPF_1), out_2 = out(CBPF_2)$
  $SF_{res} = SF_1 \cup SF_2 \setminus \{(pred(comp1), comp1), (comp1, succ(comp1)), (in_2, succ(in_2)),$
  $(pred(out_2), out_2)\} \cup \{(pred(comp1), succ(in_2)), (pred(out_2), succ(comp1))\}$

- The swimlanes, artifacts, message flow and associations of the result are the union of their counterparts from the input processes:

  $S_{res} = S_1 \cup S_2$
  $Ar_{res} = Ar_1 \cup Ar_2$
  $MF_{res} = MF_1 \cup MF_2$
  $AS_{res} = AS_1 \cup AS_2$

- The composition interface of the result contains the union of interfaces of the input models, from which we remove the tagged activity of $CBPF_1$ and the start and tagged end event of $CBPF_2$:

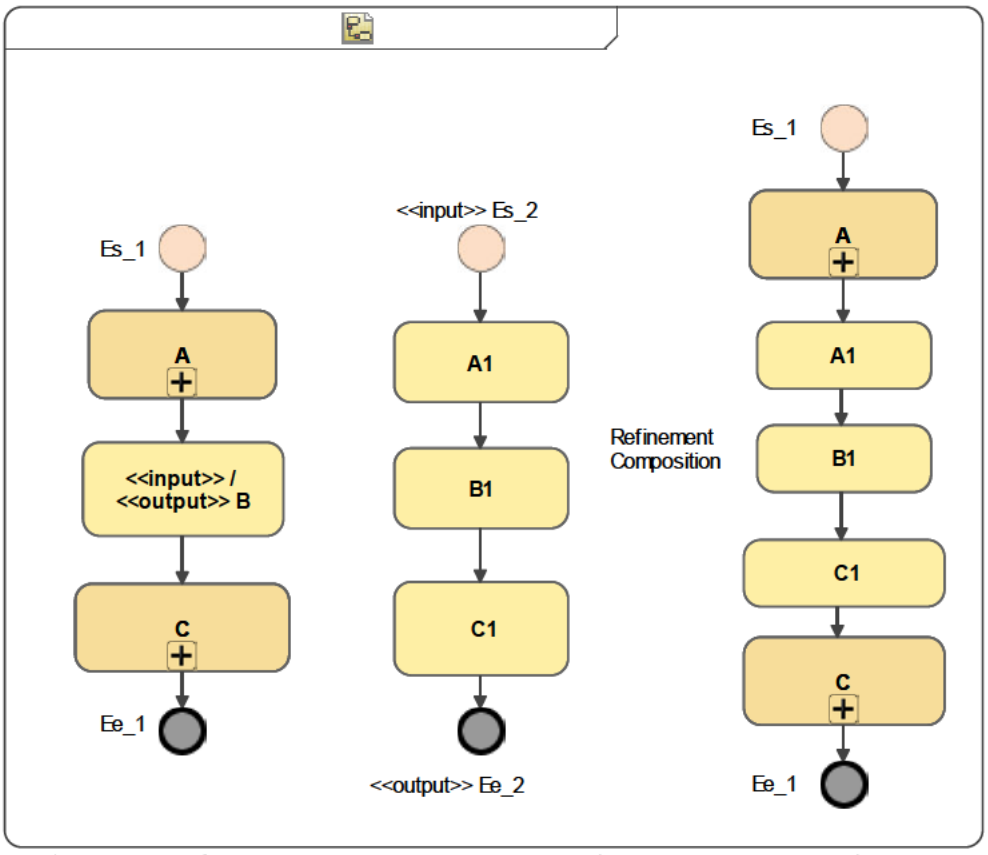


Fig. 7.7: Refinement composition operator for business process fragments

$$I_{res} = I_1 \cup I_2 \setminus \{comp1, in(CBPF_2), out(CBPF_2)\}$$

The general functioning of the operator is graphically depicted in Figure 7.7.

Synchronization composition operator:

Definition: Let $CBPF_1 = (F_1, S_1, Ar_1, SF_1, MF_1, AS_1, I_1)$ and $CBPF_2 = (F_2, S_2, Ar_2, SF_2, MF_2, AS_2, I_2)$ be two business process fragments. The result of applying the synchronization composition operator on the process fragments $(CBPF_1)$ and $(CBPF_1)$, denoted $sync(CBPF_1, CBPF_2)$, is a new business process fragment $CBPF_{res} = (F_{res}, S_{res}, Ar_{res}, SF_{res}, MF_{res}, AS_{res}, I_{res})$ where:

- To perform the composition we need to "synchronize" the activities that belong to the synchronization set Sync. This can be performed in three possible ways. For each pair of activities from Sync, we can: add a new activity that represents their synchronization; keep the first element of the pair (the activity that belongs to fragment $CBPF_1$) to represent their synchronization; keep the second element of the pair (the activity that belongs to fragment $CBPF_2$) to represent their synchronization. For ease of use, we denote by *SolvedSync* the set of activities that have been synchronized.

  The activities of the result are the union of the activities of the input process fragments, from which we need to remove the activities belonging to the synchronization set *Sync* and add the ones that represent their synchronization (one of three possible choices listed above):

$A_{res} = A_1 \cup A_2 \setminus \{(x,y)|(x,y) \in Sync\} \cup \{sync_i|sync_i \in SolvedSync\}$, where $x \in A_1, y \in A_2$;

- The events of the result are the union of events of the input models, from which we remove the start and end events and add new start and end events:

  $E_{res} = E_1 \cup E_2 \setminus \{in(CBPF_1), out(CBPF_1), in(CBPF_2), out(CBPF_2)\} \cup \{start_{new}, end_{new}\}$, where $E_{res} \subseteq F_{res}, E_1 \subseteq F_1, E_2 \subseteq F_2, start_{new} \in \mathcal{E}_s, end_{new} \in \mathcal{E}_e$

- To obtain the gateways of the result, we take the union of gateways of the input models and add several new parallel gateways; their number depends on the number of elements in the synchronization set Sync:

  $G_{res} = G_1 \cup G_2 \cup \{g_i|g_i \in \mathcal{G}_p, i = 1..2 * |Sync|\}$

- When the synchronization set Sync has only one element, it defines two fragments (above, bellow) on each input process. To obtain the sequence flow of the result, the above fragments are first composed in parallel using parallel gateways; the same applies for the below fragments. Then the results thus obtained are composed in sequence, adding between them a new element that is the merging of the synchronization elements:
  For simplicity and to improve the understanding, we use the following notations:
  $in_1 = in(CBPF_1), in_2 = in(CBPF_2), out_1 = out(CBPF_1), out_2 = out(CBPF_2)$
  $SF_{res} = SF_1 \cup SF_2 \setminus \{(in_1, succ(in_1)), (pred(out_1), out_1), (in_2, succ(in_2)), (pred(out_2), out_2), (pred(act1), act1), (act1, succ(act1)), (pred(act2_1), act2_1), (act2_1, succ(act2_1))\} \cup \{(start_{new}, g_1), (g_1, succ(in_1)), (g_1, succ(in_2), (pred(act1), g_2), (pred(act2), g_2), (g_2, act12), (act12, g_3), (g_3, succ(act1)), (g_3, succ(act2)), (pred(out_1), g_4), (pred(out_2), g_4), (g_4, end_{new}), (start_{new}, g_1)$, where $Sync = \{(act1, act2)\}, SolvedSync = \{act12\}, g_i \in \mathcal{G}_p, i = 1..4$

  If the synchronization set has more than one element ($|SyncSet| > 1$) then to obtain the sequence flow of the result SF_{res} we follow the same procedure, but this time we also have to compose the process fragment parts created between successive synchronization elements.

- The swimlanes, artifacts, message flow and associations of the result are the union of their counterparts from the input processes:

  $S_{res} = S_1 \cup S_2$
  $Ar_{res} = Ar_1 \cup Ar_2$
  $MF_{res} = MF_1 \cup MF_2$
  $AS_{res} = AS_1 \cup AS_2$

- The composition interface of the result is the union of the ones of the input process fragments, from which we remove the elements belonging to the synchronization set Sync, and add an input composition tag on the new start event and an output composition tag on the new end event:

  $I_{res} = I_1 \cup I_2 \setminus \{x, y|(x,y) \in Sync, x \in A_1, y \in A_2\} \cup \{start_{new}, end_{new}\}$, where $Tag(start_{new}) \in CT_i, Tag(end_{new}) \in CT_o$

The general functioning of the operator is graphically depicted in Figure 7.8.

Insertion composition operator:

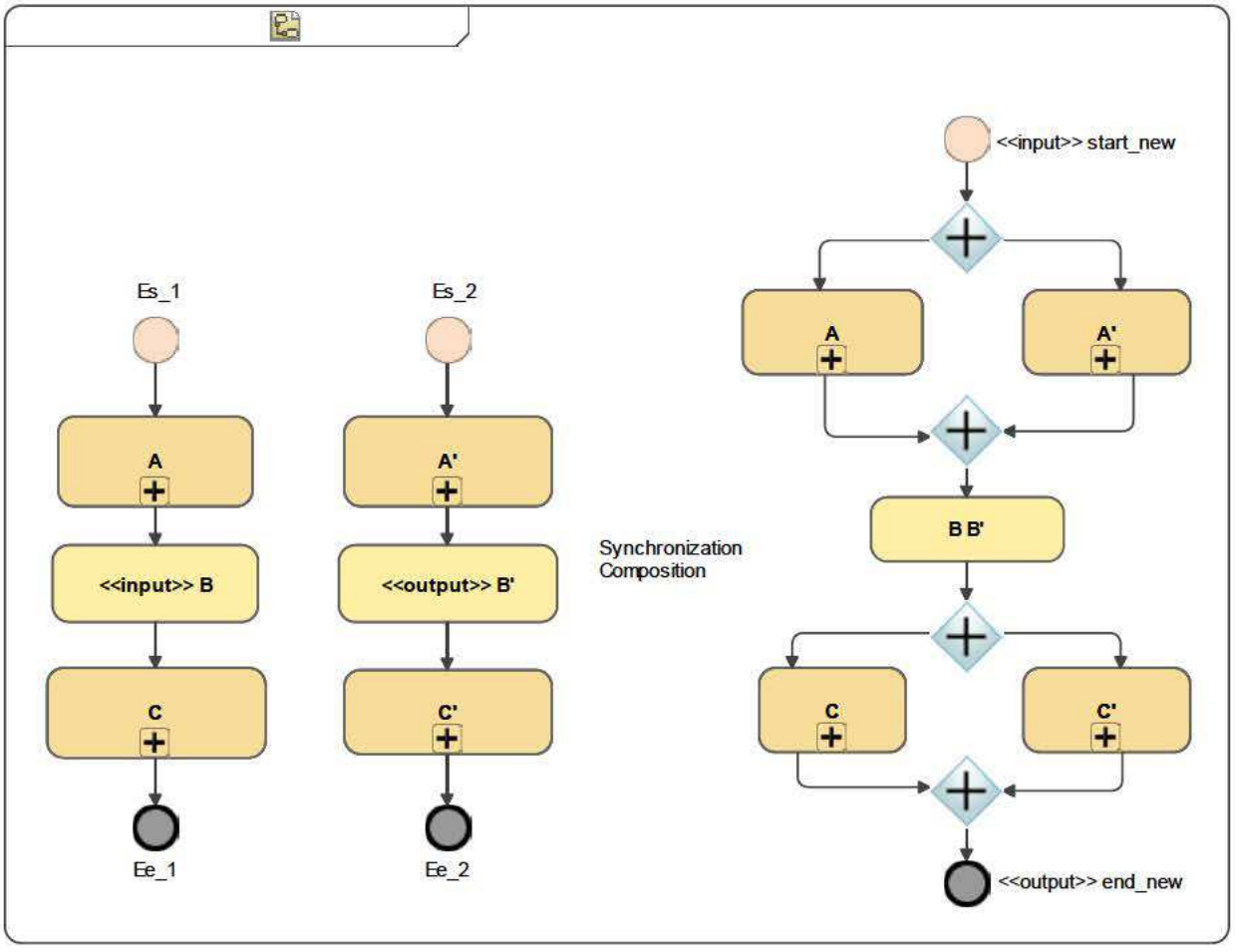


Fig. 7.8: Synchronization composition operator for business process fragments

Definition: Let $CBPF_1 = (F_1, S_1, Ar_1, SF_1, MF_1, AS_1, I_1)$ and $CBPF_2 = (F_2, S_2, Ar_2, SF_2, MF_2, AS_2, I_2)$ be two business process fragments. The result of applying the insertion composition operator on the process fragments $(CBPF_1)$ and $(CBPF_1)$, denoted $ins(CBPF_1, CBPF_2)$, is a new business process fragment $CBPF_{res} = (F_{res}, S_{res}, Ar_{res}, SF_{res}, MF_{res}, AS_{res}, I_{res})$ where:

- The activities of the result are the union of the activities of the input business process fragments:

  $A_{res} = A_1 \cup A_2$

- The events of the result are the union of events of the input models, from which we remove the start event and tagged end event of fragment $CBPF_2$:

  $E_{res} = E_1 \cup E_2 \setminus \{in(CBPF_2), out(CBPF_2)\}$

- To obtain the gateways of the result, we take the union of gateways of the input process fragments:

  $G_{res} = G_1 \cup G_2$

- For the *insert before* composition: the sequence flow of the result is the union of the sequence flows of the input process fragments, from which we remove the sequence flow connecting the tagged activity of fragment $CBPF_1$ with its predecessor and also the sequence flows connecting the start and tagged end event of fragment $CBPF_2$. We then connect, by adding sequence flow relations, the predecessor of the tagged activity from $CBPF_1$ to the first flow object of $CBPF_2$, and the last flow object of $CBPF_2$ with the tagged activity of $CBPF_1$:
  For simplicity and to improve the understanding, we use the following notations:
  $in_1 = in(CBPF_1), in_2 = in(CBPF_2), out_1 = out(CBPF_1), out_2 = out(CBPF_2)$
  $SF_{res} = SF_1 \cup SF_2 \setminus \{(pred(act1), act1), (in_1, succ(in_2)), (pred(out_2), out_2)\}$
  $\cup \{(pred(act1), succ(in_2)), (pred(out_2), act1)\}$

  For the *insert after* composition: the sequence flow of the result is the union of the sequence flows of the input process fragments, from which we remove the sequence flow connecting the tagged activity of fragment $CBPF_1$ with its successor and also the sequence flows connecting the start and tagged end event of fragment $CBPF_2$. We then connect, by adding sequence flow relations, the tagged activity from $CBPF_1$ to the first flow object of $CBPF_2$, and the last flow object of $CBPF_2$ with the successor of the tagged activity of $CBPF_1$:

  $SF_{res} = SF_1 \cup SF_2 \setminus \{(act1, succ(act1)), (in_1, succ(in_2)), (pred(out_2), out_2)\}$
  $\cup \{(act1, succ(in_2)), (pred(out_2), succ(act1))\}$

- The swimlanes, artifacts, message flow and associations of the result are the union of their counterparts from the input processes:

  $S_{res} = S_1 \cup S_2$
  $Ar_{res} = Ar_1 \cup Ar_2$
  $MF_{res} = MF_1 \cup MF_2$
  $AS_{res} = AS_1 \cup AS_2$

- The composition interface of the result is the union of those of the input process fragments, from which we remove the tagged activity of fragment $CBPF_1$ and the

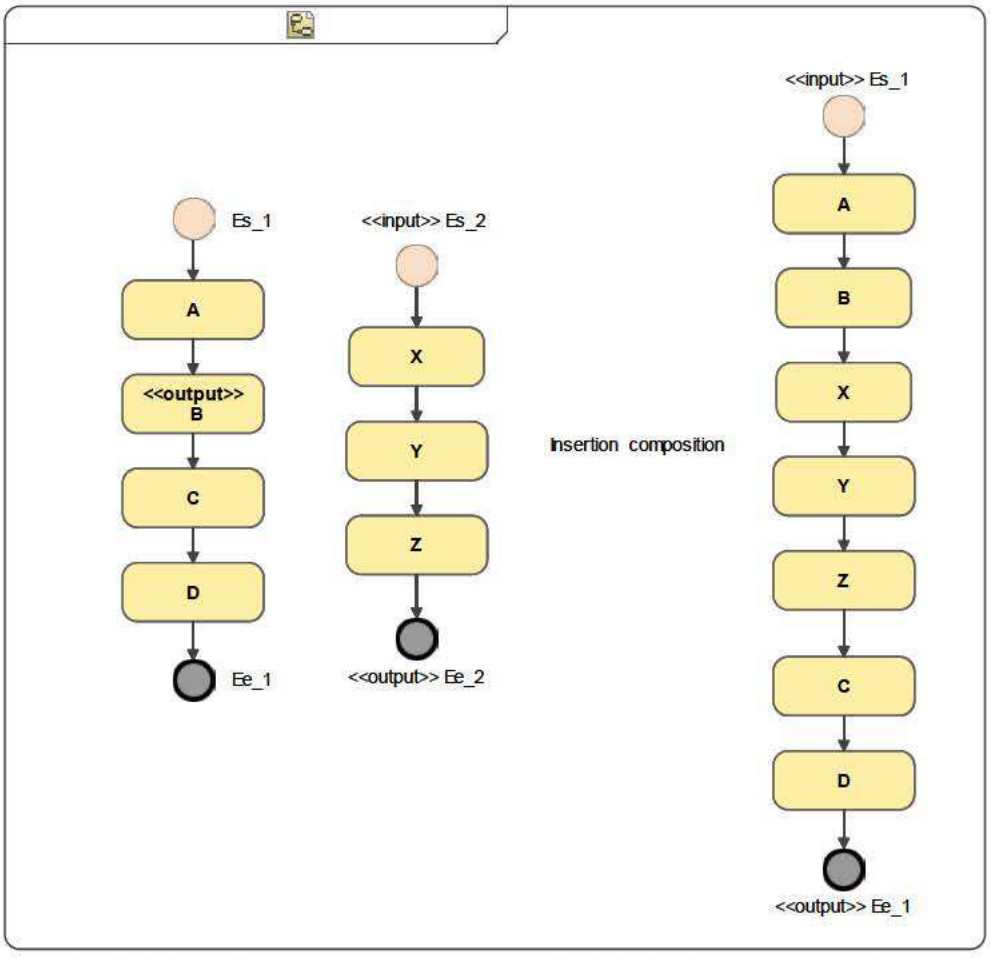


Fig. 7.9: Insert after composition operator for business process fragments

start and tagged end event of fragment $CBPF_2$. We then add an input composition tag on the start event of the result and an output composition tag on the end event:

$I_{res} = I_1 \cup I_2 \backslash \{act1, in(CBPF_2), out(CBPF_2)\} \cup \{Es_{res}, Ee_{res}\}$, where $Es_{res} = Es_1$, $Ee_{res} \in Ee_1$ and $Tag(Es_{res}) \in CT_{in}$, $Tag(Ee_{res}) \in CT_{out}$

The general functioning of the operator is graphically depicted in Figure 7.9. This concludes the specification of the CBPF composition operators.

7.2 Composing business process fragments using aspect weaving

Throughout this thesis we focused on the modelling and composition of business process fragments. In Chapter 4 we proposed the CBPF language, a domain specific language created exactly for these purposes. For the composition of business process fragments, the CBPF language proposes a set of composition operators. However, there exist other ways and methods that can enable the composition of business process fragments. We consider that concepts and principles from the field of Aspect Oriented Modelling, especially *aspect weaving*, can be of great benefit in this direction. Thus, in the following we introduce the idea of applying aspect weaving for composing business processes as an interesting research perspective and present the first steps in this direction. We acknowledge the fact that this idea should be analysed and studied in more depth.

Although composition is a central concern in both Business Process Modelling and Aspect Oriented Modelling domains, its definition is quite different in each case. While composition of business processes is a rather symmetric approach, where each view belongs to the same model, the AOM composition is asymmetric, considering one part as a base model and the other as an aspect that intend to modify the base. In many case, it would be useful to combine these two approaches. To simplify the task of the designer, it would be easier if a unique method/tool could be use to perform the composition, whether we handle two models or a model and an aspect.

The composition of behavioural models is a quite complex task, much more intricate than that of structural models. The main issue is that there exists a lot of possible composition for two processes, each of them leading to a semantically different result. We have seen that the result of composing two business processes may have different semantics, corresponding to, for example: a sequential execution of processes, a choice between the processes, a competition between the execution of processes. Each of this result can be performed by a specific composition operator propose by the CBPF language, and we could have added many other composition results to the lists. There are plenty of possible composition for two processes. Thus, composition is not restricted to a systematic operation but implies a choice between several strategies.

In this context it is useful to provide an efficient way to implement these composition operators, and to provide tools that help the designers to choose the right compositions operators for some given processes. Our proposition is to study and provide a unified approach that relies on aspect-oriented modelling concepts to implement these compositions.

Being able to use aspect weaving to compose business processes has several benefits. First, it means that you will have a single algorithm to implement the composition, which facili-

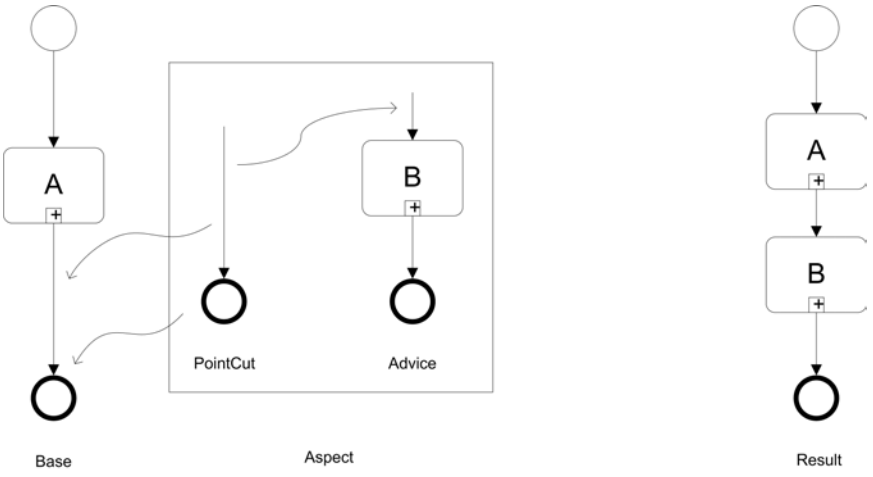


Fig. 7.10: Sequential weaving with aspects

tates the implementation of different composition strategies and the design of new compositions. This feature is very useful in the domain of business processes, as composition may have different meanings. Second, if the designer wants to be able to decompose its software into components and to use aspects, the whole composition phase can be performed by the same tools.

In the following we explain how aspect weaving can be used to perform behavioural model composition. The general idea is that two business process fragments can be composed by first transforming one of them into an aspect and then weaving it with the other business process. In Figure 7.10 we introduce a simple example that shows how two business processes can be composed to produce a new process. Our approach consist in transforming the second business process into an aspect that can be weaved with the first one to obtain the expected result. To do so, we need to define how the pointcut, the advice and the mapping from the first to the latter can be extracted from the second process.

The pointcut aims at defining what part of the first process will be transformed during the composition. It must be composed of the element that need to be replaced or removed during the composition and of the information required to identify precisely the location of the weaving. In the example, we need to indicate that we will plug the new fragment at the end of the first one. So, we need to isolate the end event of the first process in the pointcut. We also put into the pointcut the transition that leads to the end event because we need to update it after the weaving, so that it leads to the start of the second process afterwards.

The advice defines what will be inserted or removed during the weaving. Here, it is compose of all the elements the second process except the start event (that will be replaced by the last transition of the first process). Finally, the mapping between the pointcut and the advice is straightforward: the final transition of the point cut is mapped to the first transition of the advice. The result, as well as the graphical representation of the aspect, is given in Figure 7.10.

In the following, we take some of the composition operators proposed in the CBPF language and explain how they can be replaced by aspect weaving and which are the required operations for doing this. We start with the *choice composition operator*, that aims to put

6.2 Choice (or Inclusive or Alternative)

Goal:

The Choice composition aims to put a fragment in parallel to another thanks to two inclusive gateways.

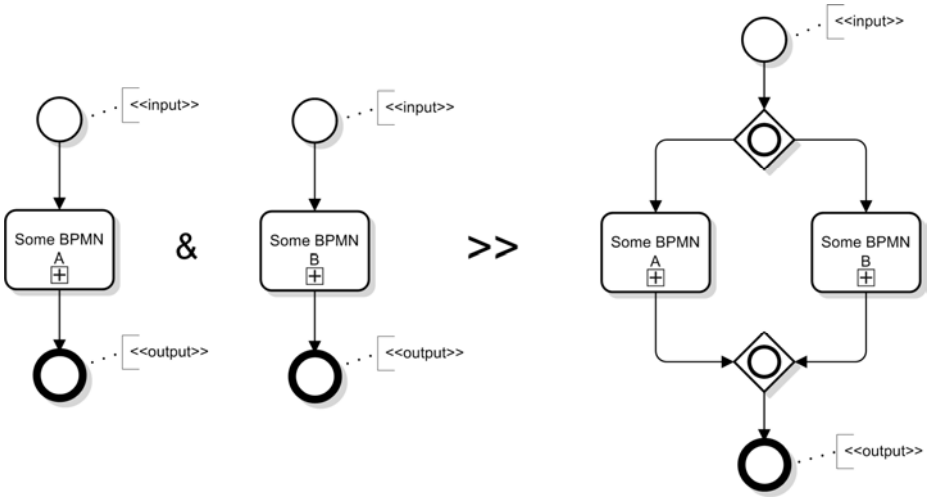


Figure 9 Choice composition

Fig. 7.11: Choice composition of business process fragments

Input:

The processor deals with an annotated BPMN fragment that has:

- a StartEvent Activity tagged with an "input" composition interface
- an EndEvent Activity tagged with an "output" composition interface

a fragment in parallel to another thanks to two inclusive gateways. An example of applying this operator is available in Figure 7.11.

In order to be able replace the choice composition operator by an aspect weaving approach, we first need to choose which one of the two business process fragments is considered as *base model* for the weaving. This fragment will not be altered and no further processing needs to be performed on it. Implicitly, the other business process fragment, which was not selected as base, need to be transformed in an *aspect*. This means that we need to build the appropriate *advice* and *pointcut*. The construction of the aspect is an operation that highly depends on the type of composition that we want to perform. In other words, a business process fragment will be transformed differently into an aspect, depending on which type of composition we want to apply.

In the case of the choice composition, a business process fragment is transformed into an aspect in the following manner:

- *Create the advice:* we start from the original business process fragment that needs to e transformed. We take the start event, tagged with an input composition interface, and add a *splitting OR gateway* after it. On one of the outgoing branches of this gateway, we add all of the flow objects of the initial business process fragment. The other branch will only contain a simple sequence flow with nothing connected to it. Both branches converge into a *merging OR gateway*, which is then connected by an outgoing sequence flow to a tagged end event.

- *Create the pointcut:* we create a new business process diagram that contains an *EndEvent* tagged with an output composition interface, then add a "floating" *SequenceEdge* that has no source but has the *EndEvent* as target and put on it the same g-morphism identifier as used in Advice for the "floating" *SequenceEdge* connected to the gateway connected to the *EndEvent*. We do the same for *StartEvent*.

To facilitate the understanding, we present this process in Figures 7.12 and 7.13. Figure 7.12 presents a simple generic business process fragment used as input for the transforma-

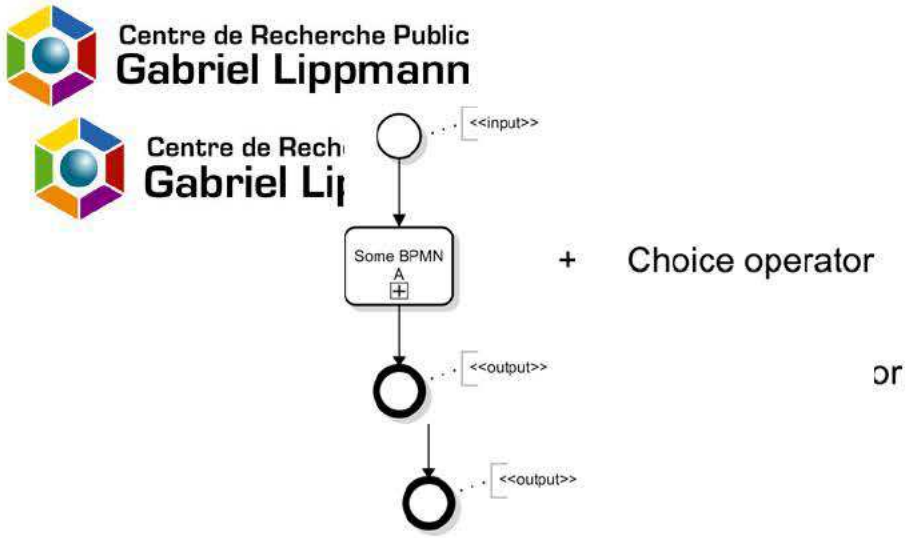


Fig. 7.12: Initial fragment and choice operator

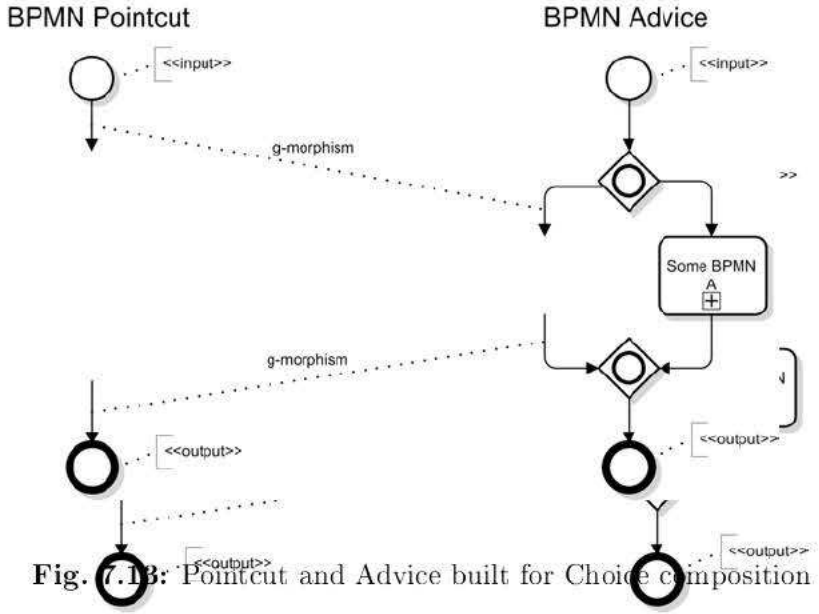


Fig. 7.13: Pointcut and Advice built for Choice composition

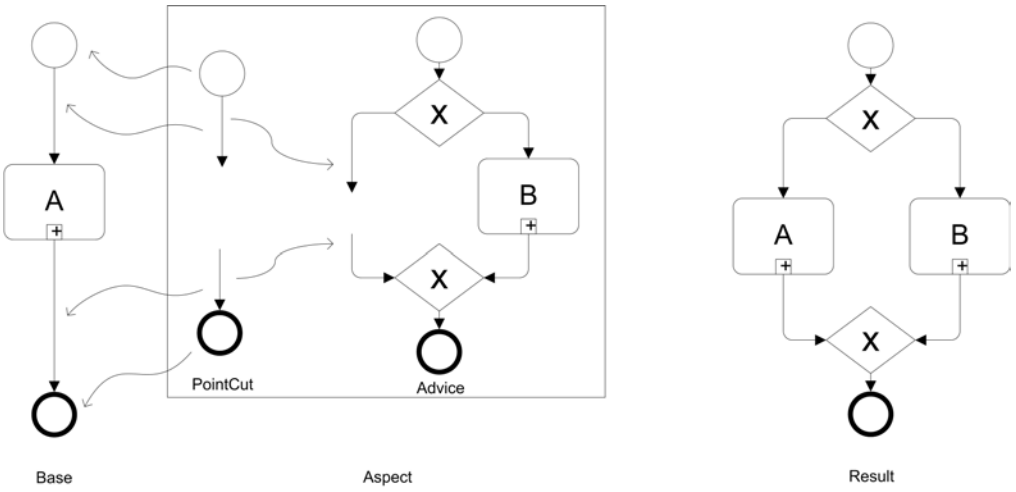


Fig. 7.14: Exclusive choice composition of processes with aspects

tion and mentions the fact that the choice composition will be used. In Figure 7.13 we can see the pointcut and advice that are created.

In the same manner as presented above, we can replace the exclusive choice composition operator by a specific aspect weaving. In Figure 7.14 we present how a business process fragment can be transformed into an aspect for this specific type of composition (creation of the pointcut and advice) and how to obtain the result of the weaving.

The procedure presented in Figure 7.14 works well as long as we want to run in parallel all the content of the two business processes. However, often we will have to integrate a choice only for a part of one of the two business processes. To identified the part of the process that will be taken into account for the choice operation, composition annotations can be introduced: an annotation *c-start*, respectively *c-end*, will identify the beginning (resp. the end) of the choice section. In this case, applying the choice composition using aspect weaving is more intricate. We consider the following three criteria:

- Does the sub-process that will be integrated to the choice part starts the whole process?
- Does the sub-process that will be integrated to the choice part ends the whole process?
- Is there more than one element in that process?

These criteria are independent. Thus, it leads to a total of eight possibilities. Unfortunately, each one corresponds to a particular aspect. These aspects are provided in Figure 7.15. The four on the left are for process fragments that are composed of a single element while those on the right are composed of several elements. The two of the first line have neither any element before or after the business process part to set in parallel. The two of the second line have elements before the business process part to set in parallel, but none after. On the third line, there are elements after the business process part to set in parallel but none before. On the last line, there are elements before and after the business process part to set in parallel.

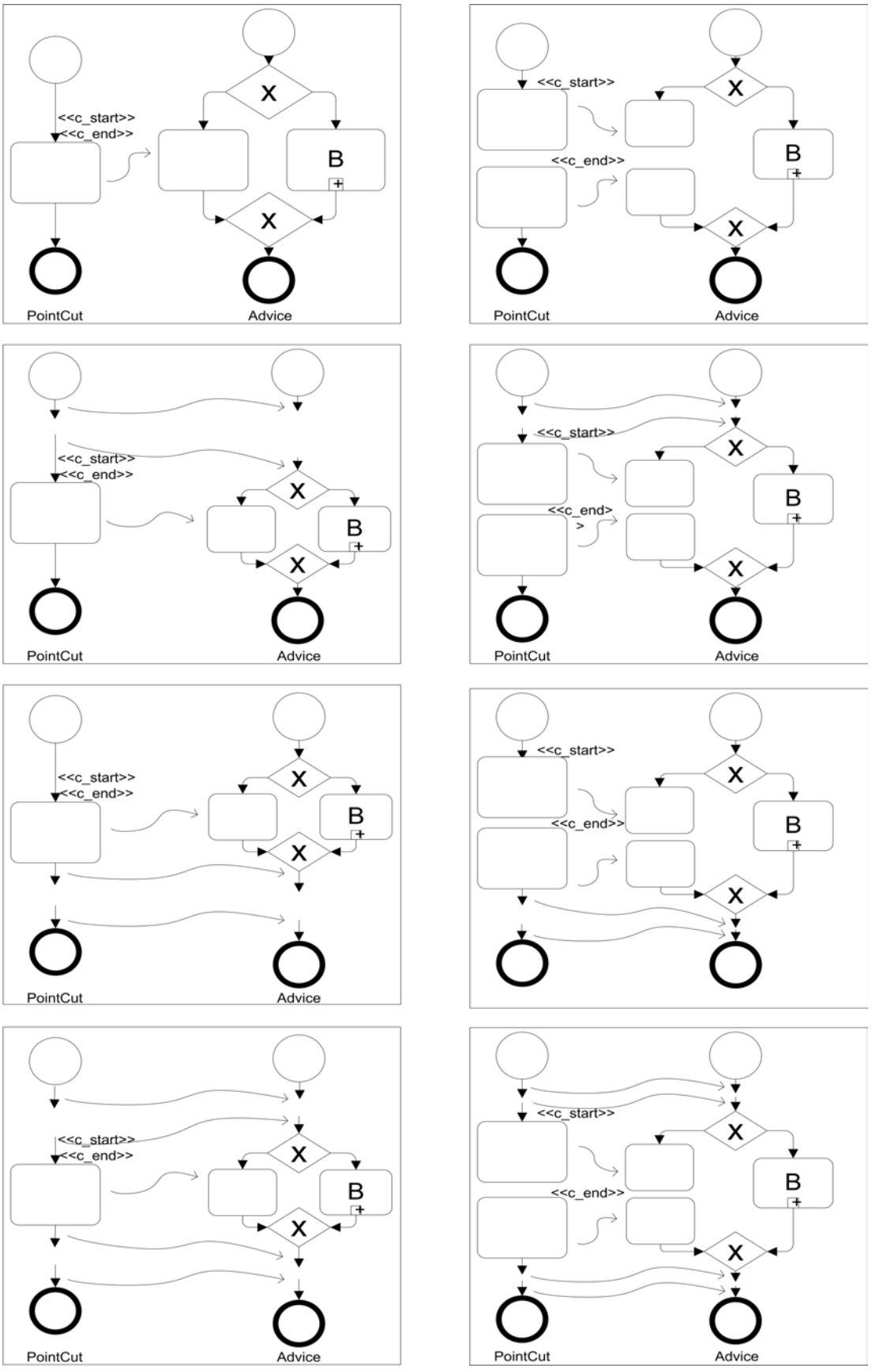


Fig. 7.15: The eight aspects for the choice composition

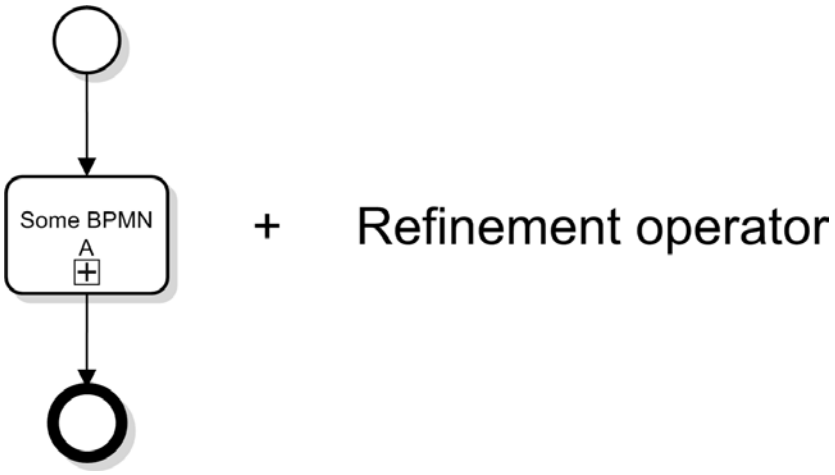


Figure 7.16 Input for the refinement composition operator

Figure 19 Refinement processor input

As it can be seen, each aspect as a distinct pointcut and the pointcuts are not ambiguous. For two given annotation c start and c end, it exists at most one pointcut out of the eight of Figure 7.15 such that there is a joinpoint that includes that includes these annotations. Moreover, the start event Activity and the end event process activity is only one applicable aspect.

Another example of how to replace a composition operator by a specific aspect weaving technique can be given for the *refinement composition operator*. This composition simply aims to replace a task of one business process fragment with an entire business process fragment, in order to give more detail about that activity. In order to be able to apply this type of composition, we need to explicitly mark on one of the business process fragments the exact task that has to be refined. This is done by using input or output composition tags.

In order to transform a business process fragment into an aspect specific for this type of composition, the following operations need to be performed:

- *Build the Advice:* we need to remove the *StartEvent* and the *EndEvent* from the business process fragment. We also tag the two remaining "floating" SequenceEdges with two new g-morphism identifiers.
- *Build the Pointcut:* the processor creates a new business process diagram, adds a Task, adds an "input" composition interface on it, adds a "floating" SequenceEdge that has no source but has the Task as target and put on it the same g-morphism identifier as used in Advice on the SequenceEdge that as no source.

Figure 7.16 graphically presents the input for the refinement composition process. Then in Figure 7.17 we can see how the business process fragment is converted into an aspect specific for the refinement composition.

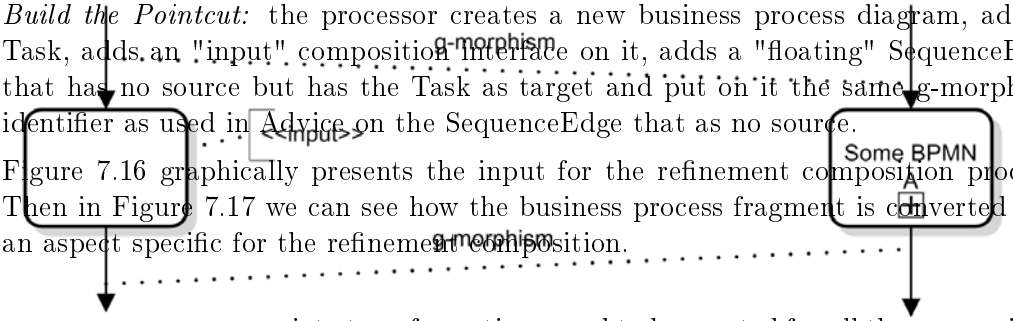


Figure 20 Pointcut1 and Advice built by Refinement processor

In the same manner, appropriate transformations need to be created for all the composition operators defined in CBPF, in order to implement them using aspect weaving methods. As we have seen above in the case of the exclusive choice operator, many complicated cases might appear and thus the transformations form fragment to aspect require some

ld the Advice:

e processor removes the StartEvent Activity and the EndEvent Activity. And tags the t
haining "floating" SequenceEdge with two new g-morphism identifiers.

ld the Pointcuts:

ntcut 1: the processor creates a new BPMN diagram, adds a Task, adds an "input"
mposition interface on it, adds a "floating" SequenceEdge that has no source but has
sk as target and put on it the same g-morphism identifier as used in Advice on the
quenceEdge that as no source.

ntcut 2: processor does the same as previously but adds an "output" composition in
the created Task instead of an "input" one.

tput:

t 1: the processor creates a new BPMN diagram, adds a Task, adds an “input” ition interface on it, adds a “floating” SequenceEdge that has no source but has the target and put on it the same g-morphism identifier as used in Advice on the ceEdge that as no source.

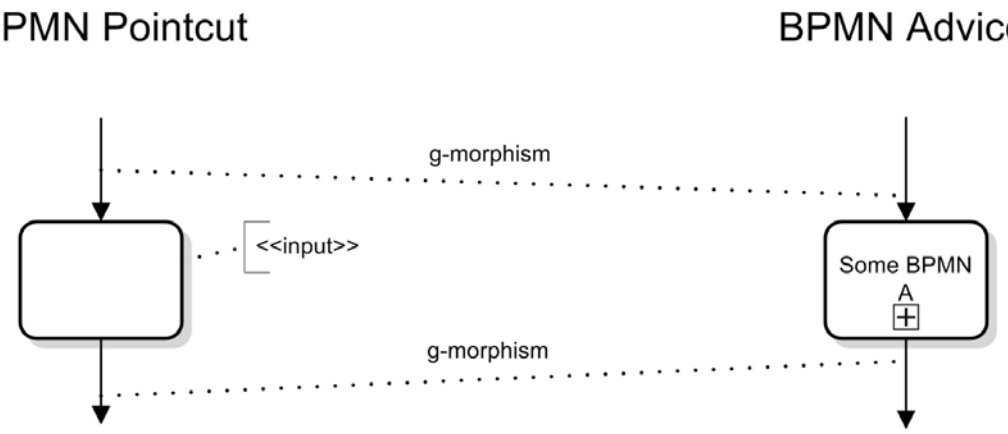


Figure 20 Pointcut1 and Advice built by Refinement processor

Fig. 7.17: Pointcut1 and Advice built for Refinement composition

t 2: processor does the same as previously but adds an “output” composition interfa created Task instead of an “input” one.

effort. However, one this is done, the obvious advantage is that a unique composition technique can be applied in all cases for all the composition types. Any of the classical aspect model weaving techniques defined by the AOM research community can be applied. Moreover, another advantage of this approach is the fact that new types of compositions can easily be added, without any modification required to the aspect weaver which remains the same. To add a new composition, we simply need to define the fragment to aspect transformation. Taking all of this into account, we consider this topic to be of great interest and an interesting possible research direction for the future.

pects due to the two created pointcuts.

Parallel with communication

7.3 Modelling data for business process fragments

rallel with communication composition aims to put a fragment in parallel to another to two parallel gateways and adds one or more message communications between Tasks.

Data representation is an important part of any business process modelling language. Therefore, it is also of the utmost importance to propose data modelling for the CBPF language for business process fragments. Many times, when executing a business process fragment, there may be data produced, either during or after the end of the process. A traditional requirement of process modelling is to be able to model the items (physical or information items) that are created, manipulated, and used during the execution of a process fragment.

In its current status, the CBPF language provides a minimal set of concepts for data modelling as part of the CBPF abstract syntax. *Data objects*, *data specifications* and *data associations* are used for representing data and data flow in a business process fragment. These concepts were introduced in Section 4.2.2.

We proposed *data objects* as a mechanism to show how data is required or produced by the activities of a business process fragment. Data objects are introduced in the language as specific types of artifacts. Activities often required data in order to execute. In addition, they may produce data during or as a result of execution. Therefore, we propose two types of data objects: *data input* and *data output*. *Data associations* are a specific type of connecting object. They are used to associate data objects with ow objects. Associations are needed to show the data inputs and outputs of activities. Finally, every data object will have a unique *type*. We propose three elementary data-types: *IntObject* (represents

integer data objects); *StringObject* (represents string data objects); *BoolObject* (represents boolean data objects).

However, this type of data representation is a very basic one and is very limited in the type of information that it provides to the user regarding the overall data flow of the business process fragment. Moreover, it limits the types of data analysis that may performed on a business process fragment. Therefore, we consider that an extension if the CBPF language with a set of concepts that allow an in-depth representation of data for a business process fragment is a promising research direction for the future.

A first improvement that can be brought is to re-think which is the most appropriate way in which data can be represented in a business process fragment. For the moment, we make use of specific data objects. However, this solution might be too rigid and restrictive. One possibility if to make use of *meta-class attributes* for representing data in the CBPF meta-model. This solution could prove to be more flexible.

Another interesting research idea is to see how data objects are related to the flow objects of a business process fragments. Moreover, for the moment we only associate data objects to the tasks of a business process fragment. In the future, we plan to study how data can be added to some of the other flow objects of a business process fragment, like gateways or event triggers. This extension of data representation to more elements of a business process fragment would facilitate the accurate modelling of the data flow over the entire fragment.

Another point that can be improved concerns the types of data that can be modelled in a business process fragment. For the moment, CBPF provides only a minimalistic set of data types which cover the basic data representations (string, integer and boolean). It is mandatory to enrich this set of data types that can be offered by the language. Many more basic data types could be added. Moreover, an important improvement could be brought by introducing compound data types. Such data types, which could be inspired by those that already exist in most programming languages, facilitate the representation of complex and structured data. Even a data type hierarchy could be envisioned in such a case as a possible language extension.

The appropriate representation of data is only the first step towards the complete modelling of data flow in a business process fragment. Therefore, once the above-mentioned ideas are put into practice and this the CBPF language extended, any language user will have access to the data flow of the business process fragment that he is creating. Implicitly, questions regarding data dependencies in a business process fragment will need to be investigated in more detail.

A proper representation of the data flow also opens the door for a series of data flow analysis techniques that may be applied on business process fragments. For the moment, the verification of business process fragments does not involve any data related properties and queries. However, we plan to extend the type of analysis that may be applied on a business process fragment and propose a set of new data flow related properties that could be studied. This will allow the user to gain even more insight into the process that he is creating or using and increase the degree of certainty that the business process behaves in a correct manner.

8. CONCLUSION

Abstract

This chapter outlines the main research questions that we propose to solve in this thesis. It also states the major contributions of the work presented in this dissertation and draws some conclusions about it.

Software Product Line Engineering is a recent software development paradigm that offers software suppliers/vendors new ways to exploit the existing *commonalities* in their software products and to support a *high level of reuse*, thus generating important quantitative and qualitative gains in terms of productivity, time to market, product quality and customer satisfaction. This technique has gained a lot of attention in recent years by both research and industry.

This thesis investigates how recent software engineering breakthroughs such as *Model Driven Engineering* and *Business Process Modelling* can be combined to devise a new and improved software product line engineering methodology. More specifically, our research was driven by the increasing need that arises in the SPL research field and community for new product derivation techniques. Further more, we have noticed that most of the work that advocates the use of model driven engineering techniques for software product line engineering addresses only the derivation of structural product representations, neglecting or just briefly addressing the problems inherent to the *derivation of product behaviour*. This yields an unwanted situation, as the behavioural product representation is as important as the structural one. Moreover, the few existing techniques that try to address to some extent the issue of derivation of product behaviour in a software product line lack the "end-to-end" dimension, which means that they do not cover both domain engineering and application engineering phases of the SPLE process.

Therefore, the main motivation and initial driver of this thesis is the study and improvement of software product line engineering methodologies with the focus on behavioural product derivation. Accordingly, the major research problem addressed throughout this thesis and thus our main claimed contribution is *the definition of a new software product line engineering methodology that covers both domain engineering and application engineering phases of the SPLE process and which focuses on the derivation of behavioural models of SPL products*. We impose the condition that the behavioural models obtained as a result of applying this methodology should describe the business and operational step-by-step workflows of activities/actions performed by the derived product. Moreover, we require several qualities from the proposed methodology: easily maintainable, scalable, comprehensible, suitable, expressive enough and to easily support modifications. We also want to develop this methodology following model driven engineering principles, as they allow to reduce design complexity and make software engineering more efficient by shifting the focus from implementation to modelling.

Several other inherent research challenges emerge and need to be solved in order to provide a proper solution to the main research question that we try to answer in this thesis:

- As the main focus of the methodology is to obtain behavioural representations of SPL products, another related problem that needs to be solved is: *how to model a complex behaviour starting from several simpler ones?* One of the factors that contributes to the difficulty of developing complex behaviours is the need to address multiple concerns in the same artefact.
- Another challenge lies ahead is to find both the adequate behavioural formalism that fits the needs of the analyst as well as a formal composition mechanism that facilitates the generation of the expected behavioural model.
- Regarding the actual composition of business processes, there are currently only a few proposals. This is currently very much a manual activity, which requires specific knowledge in advance and takes up much time and effort. The composition problem is one that cannot easily be solved by a "copy and paste" approach, as it may introduce problems like: redundancy, update anomalies or inconsistent behaviour in the resulting models. There is also a need for a formal foundation and notation for the compositions which allows the creation of business process models from model fragments.
- From a practical and technological perspective, another possible challenge is to propose and deliver the appropriate tool support for the methodology. The availability of good tool support will enable users to better understand and more easily apply the proposed methodology.

As an answer to the main research question that this dissertation tries to address, we introduced in Chapter 3 the main contribution of this thesis: a new software product line engineering methodology that focuses on the derivation of product behaviour. By applying this methodology, we can produce behavioural models that belong to the analysis and early design levels of the software system development life-cycle. The proposed methodology covers only the derivation of behavioural product models and does not address the structural product representation. However, it can be used together with other product derivation techniques for obtaining the structural product models. The methodology follows the classical SPLE process and covers both Domain Engineering and the Application engineering phases:

- During domain engineering, we capture domain knowledge using the newly proposed concept of *composable business process fragments*, which are our core assets base from which new behavioural product models will be later created. We choose to capture the commonality and variability in the domain in a separate variability model, represented as a *feature diagram*. We apply the separation of concerns principle and keep the core assets and the variability representations separate. Moreover, in order to facilitate the product derivation process, we connect features from the feature diagram to business process fragments by association. Moreover, we want to ensure that the created business process fragments are correct prior to composition. Thus, we propose to apply a new business process fragment correctness verification approach on our core assets.
- During application engineering, we create new products from the core assets base using a compositional approach. Our derivation approach uses positive variability to creates a new business process that models the behaviour of the derived product. In a first step, we require the contribution of the user for creating a particular

product configuration based on a selection of features. The business process fragments associated to the selected features are also implicitly selected. We then create a composition workflow that explicitly defines both the order in which the selected fragments are composed and also the composition operators that will be applied. The composition operators that we propose are used for composing the business process fragments, resulting the final behavioural product representation.

In Chapter 4 we proposed a new domain specific language called *Composable Business Process Fragments (CBPF)*, created specifically as the necessary language support for the proposed methodology. The language is based on the concept of *business process fragment*, which are the core assets used by our SPL methodology. We propose this new concept as a reusable granule for business process design that can allow for reuse of process logic. Business process fragments are designed to implement a set of requirements and model a single abstract functionality. Thus, the CBPG language allows the modelling of composable business process fragments. The language was constructed following an MDE approach. We start by defining the abstract syntax of the language by means of a meta-model, representing in an abstract way the concepts and constructs of the modelling language. We continue the language definition by proposing a unique graphical concrete syntax for the language. Finally, we define the semantics of the CBPF language following a translational approach. The CBPF language is created in an incremental manner. Initially, the language simply offers the necessary concepts for modelling business process fragments. As a solution to the problem of business process composition which we identified, we extend the CBPF language with new concepts for creating "composable" business process fragments. Moreover, we propose a set of composition operators created specifically for composition such business process fragments. Finally, as CBPF is meant to be the language support for our SPL methodology, we further extend the language with a set of concepts that allow the modelling of composition workflows and product derivation specifications.

In order to verify the correctness of the business process fragments created using the CBPF language and also to support one of the steps of our SPL methodology, we propose in Chapter 5 a verification techniques that checks the structural and behavioural correctness. The structural verification is performed by defining a set of adequate fragment consistency rules that should be valid for every business process fragment that can be created with the CBPF language. To check the dynamic behaviour of business process fragments, we first need to transform business process fragment into equivalent HCPN with the help of the model-to-model transformation that we propose. Once this is done, we take advantage of the large array of analysis and verification techniques and tools available for Petri nets and describe how to verify several generic and fragment specific behavioural properties.

It was mentioned that one of the challenge faced was to deliver the appropriate tool support for the methodology. Thus, in Chapter 6 we present the SPLIT tool suite, which is the tool support that we propose for our SPL methodology. We start by describing the general requirements that such a tool should fulfil. We then present the general architecture of the proposed tool. The SPLIT tool suite provides a practical implementation of the proposed methodology. The tool has been developed as a set of Eclipse plug-ins which are meant to be integrated together. The tool is also highly modular, so we also discuss in more details the different tool modules and the functionalities each of them provides.

BIBLIOGRAPHY

[ABM00] Colin Atkinson, Joachim Bayer, and Dirk Muthig. Component-based product line development: the kobra approach. In *Proceedings of theFirst Software Product Line Conference*, 2000.

[ACP+11] Mathieu Acher, Anthony Cleve, Gilles Perrouin, Patrick Heymans, Charles Vanbeneden, Philippe Collet, and Philippe Lahire. Reverse engineering architectural feature models. In *5th European conference on Software architecture*, pages 220–235, 2011.

[ASO05] Ruth Sara Aguilar-Savenn and Jan Olhager. Integration of product, process and functional orientations: Principles and a case study. In *APMS*, pages 375–389, 2005.

[ATK09] Sven Apel, Salvador Trujillo, and Christian Kastner. Model superimposition in software product lines. In *2nd International Conference on Theory and Practice of Model Transformations*, pages 4–19, 2009.

[ATL] ATLAS. Atlas model management architecture. http://wiki.eclipse.org/AMMA.

[Bat07] Don Batory. Program refactoring, program synthesis, and model-driven development. In *16th international conference on Compiler construction*, pages 156–171, 2007.

[BC05] Felix Bachmann and Paul Clements. Variability in software product lines. Technical Report cmu/sei-2005-tr-012, Software Engineering Institute, Pittsburgh, USA, 2005.

[BCH+10] Quentin Boucher, Andreas Classen, Patrick Heymans, Arnaud Bourdoux, and Laurent Demonceau. Tag and prune: a pragmatic approach to software product line implementation. In *IEEE/ACM international Conference on Automated Software Engineering*, pages 333–336, 2010.

[BCTS06] David Benavides, Antonio Ruiz Cortés, Pablo Trinidad, and Sergio Segura. A survey on the automated analyses of feature models. In *JISBD*, pages 367–376, 2006.

[BDC92] Luca Bernardinello and Fiorella De Cindio. A survey of basic net models and modular net classes. *Advances in Petri Nets*, 609:304–351, 1992.

[Bez04] Jean Bezivin. In search of a basic principle for model driven engineering. *Novatica Upgrade*, 2:21–24, 2004.

[Bez05] Jean Bezivin. On the unification power of models. *Software and Systems Modeling*, 4:171188, 2005.

[BF06] Thorsten Blecker and Gerhard Friedrich. *Mass Customization - Challenges and Solutions.* Springer, 2006.

[BFG00] Joachim Bayer, Oliver Flege, and Cristina Gacek. Creating product line architectures. In *IW-SAPF*, pages 210–216, 2000.

[BFG+02] Jan Bosch, Gert Florijn, Danny Greefhorst, Juha Kuusela, J. Henk Obbink, and Klaus Pohl. Variability issues in software product lines. In Springer-Verlag, editor, *4th International Workshop on Software Product-Family Engineering*, pages 13–21, 2002.

[BG01] Jean Bezivin and Olivier Gerbe. Towards a precise definition of the omg/mda framework. In *16th IEEE international conference on Automated software engineering.* IEEE Computer Society, Washington, DC, USA, 2001.

[BH02] Luciano Baresi and Reiko Heckel. Tutorial introduction to graph transformation: A software engineering perspective. In *Proceedings of the First International Conference on Graph Transformation*, page 402429, 2002.

[BIJ06] Alan W. Brown, Sridhar Iyengar, and Simon Johnston. A rational approach to model-driven development. *IBM Systems Journal*, 45:463480, 2006.

[BKR03] J. Becker, M. Kugeler, and M. Rosemann. *Process Management. A Guide for the Design of Business Processes.* Springer-Verlag: Berlin, 2003.

[Bos00] Jan Bosch. *Design and use of software architectures: adopting and evolving a product-line approach.* Addison-Wesley, 2000.

[BSW04] Stefan Berndes, Alexander Stanke, and Kai Worner. Business process modelling. In *Business Process Management for Open Processes: Method and Tool to Support Product Development Processes.* Springer, 2004.

[CA05] K. Czarnecki and M. Antkiewicz. Mapping features to models: a template approach based on superimposed variants. In *GPCE*, pages 422–437, 2005.

[Cai00] Xia Cai. Component-based software engineering: technologies, development frameworks, and quality assurance schemes. In *7th Asia-Pacific Software Engineering Conference*, pages 372 – 379, 2000.

[CCG09] Benoit Combemale, Xavier Cregut, and Xavier Garoche, Pierre-Loic andThirioux. Essay on semantics definition in mde - an instrumented approach for model verification. *Journal of Software*, 4:943–958, 2009.

[CCG+11] Alfredo Capozucca, Betty Cheng, Geri Georg, Guelfi, Paul Istoan, and Gunter Mussbacher. Requirements definition document for a software product line of car crash management systems. Technical Report CS-11-105, Computer Science Department Colorado State University, June 2011.

[CCGI11] Alfredo Capozucca, Betty Cheng, Nicolas Guelfi, and Paul Istoan. Oo-spl modelling of the bcms case study. Technical Report TR-LASSY-11-14, University of Luxembourg, 2011.

[CE00] Krysztof Czarnecki and Ulrich Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[CH03] Krzysztof Czarnecki and Simon Helsen. Classification of model tranformation approaches. In *OOPSLA Workshop on Generative Techniques in the Context of Model Driven Architecture*, 2003.

[CH06] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45:621645, 2006.

[CHE05a] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10:7–29, 2005.

[CHE05b] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. Staged configuration through specialization and multilevel configuration of feature models. *Software Process: Improvement and Practice*, 10:143–169, 2005.

[CHW98] James Coplien, Daniel Hoffman, and David M. Weiss. Commonality and variability in software engineering. *IEEE Software*, 15:37 – 45, 1998.

[CJ01] M. Clauss and I. Jena. Modeling variability with uml. In *GCSE Young Researchers Workshop*, 2001.

[CJ03] John Clark and Jeremy Jacob. Model-driven development. *IEEE Software*, 20:14–18, 2003.

[CKO92] Bill Curtis, Marc I. Kellner, and Jim Over. Process modelling. *Communications of the ACM*, 35:75–90, 1992.

[Cla01] M. Clauss. Generic modeling using uml extensions for variability. In *OOPSLA*, 2001.

[CN01] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison Wesley, 2001.

[Coa99] Workfow Management Coalition. Terminology and glossary document number wfmc-tc-1011. http://www.wfmc.org/Download-document/WFMC-TC-1011-Ver-3-Terminology-and-Glossary-English.html, 1999.

[CSW08] Tony Clark, Paul Sammut, and James Willans. *Applied metamodelling: a foundation for language driven development*. Ceteva, Sheffield, 2008.

[Dav87] S. M. Davis. *Future Perfect*. Addison-Wesley, 1987.

[Dav03] James Davis. Gme: the generic modeling environment. In *OOPSLA Companion*, page 8283, 2003.

[DdL03] Jerome Delatour and Florent de Lamotte. Argopn: a case tool merging uml and petri nets. In *NDDL/VVEIS*, pages 94–102, 2003.

[DEA03] Guy Fitzgerald David E. Avison. Where now for development methodologies? In *Communications of the ACM*, volume 46, pages 78 – 82. ACM New York, NY, USA, 2003.

[DFF+10] Zoe Drey, Cyril Faucher, Franck Fleurey, Vincent Mahe, and Didier Vojtisek. Kermeta language reference manual, November 2010.

[DGR10] Deepak Dhungana, Paul Grunbacher, and Rick Rabiser. The dopler metatool for decision-oriented variability modeling: a multiple case study. *Automated Software Engineering*, 18:77–114, 2010.

[DH00] B. Rumpe D. Harel. Modeling languages: Syntax, semantics and all that stuff, part i: The basic stuff. Technical report, Weizmann Science Press of Israel, 2000.

[dLV02] Juan de Lara and Hans Vangheluwe. Atom3: A tool for multiformalism and meta-modelling. In *FASE*, 2002.

[DMW99] C. T. R. Lai D. M. Weiss. *Software product-line engineering: a family-based software development process.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[dOJdSGHM05] Edson Alves de Oliveira Junior, Itana Maria de Souza Gimenes, Elisa Hatsue Moriya Huzita, and Jose Carlos Maldonado. A variability management process for software product lines. In *CASCON*, pages 225–241, 2005.

[DRR04] J. Desel, W. Reisig, and G. Rozenberg. *Advances in Petri Nets.* Springer, 2004.

[DS90] T. H. Davenport and J. E. Short. The new industrial engineering: information technology and business process redesign. *Sloan Management Review*, 31:11–27, 1990.

[DSB05] Sybren Deelstra, Marco Sinnema, and Jan Bosch. Product derivation in software product families: a case study. *Journal of Systems and Software*, 74:173194, 2005.

[dSMBB10] Marcos Aurelio da Silva, Alix Mougenot, Xavier Blanc, and Reda Bendraou. Towards automated inconsistency handling in design models. In *22nd International Conference on Advanced Information Systems Engineering*, volume 6051. Springer, 2010.

[DV02] Peter Domokos and Daniel Varro. An open visualization framework for metamodel-based modeling languages. *Electr. Notes Theor. Comput. Sci.*, 72:69–78, 2002.

[EBB05] Magnus Eriksson, Jurgen Borster, and Kjell Borg. The pluss approach - domain modeling with features, use cases and use case realizations. In *SPLC*, pages 33–44, 2005.

[Ecl12] Eclipse. Eclipse/ecore. http://www.eclipse.org/modeling/emft/?project ecoretools, 2012.

[EN10] Rik Eshuis and Alex Norta. Business process composition. *Dynamic Business Process Formation for Instant Virtual Enterprises*, pages 93–111, 2010.

[Fav04] Jean M. Favre. Towards a basic theory to model model driven engineering. In *Workshop on Software Model Engineering*, 2004.

[Fav05] Jean-Marie Favre. Foundations of meta-pyramids: Languages vs. metamodels episode ii: Story of thotus the baboon. In Jean Bezivin and Reiko Heckel, editors, *Dagstuhl Seminar Proceedings*, 2005.

[FB05] Frederic Fondement and Thomas Baar. Making metamodels aware of concrete syntax. In *Lecture Notes in Computer Science : Model Driven Architecture Foundations and Applications*, volume 3748 of *Lecture Notes in Computer Science*, page 190204. Springer, 2005.

[FBJ+05] Marcos Didonet Del Fabro, Jean Bezivin, Frederic Jouault, Erwan Breton, and Guillaume Gueltas. Amw: a generic model weaver. In *Procs. of IDM05*, 2005.

[FEBF06] Jean-Marie Favre, Jacky Estublier, and Mireille Blay-Fornarino. *Lingenierie dirigee par les modeles, au-dela du MDA*. Hermes Science, Lavoisier, 2006.

[FK05] William B. Frakes and Kyo Kang. Software reuse research: Status and future. *IEEE Transactions on Software Engineering*, 31:529–536, 2005.

[Fow09] Martin Fowler. Language workbenches and model driven architecture. http://martinfowler.com/articles/mdaLanguageWorkbench.html, 2009.

[GBLNJ10] Marie Gouyette, Olivier Barais, Jerome Le Noir, and Jean-Marc Jezequel. Managing variability in multi-views engineering. In *Journee Lignes de Produits*, October 2010.

[GEA] GEARS. Biglever. http://www.biglever.com/index.html.

[GFdA98] M. L. Griss, J. Favaro, and M. d Alessandro. Integrating feature modeling with the rseb. In *ICSR*, 1998.

[Gib94] Wayt Gibbs. Softwares chronic crisis. *Scientific American*, 3:72–81, 1994.

[GKT93] Subashish Guha, William J. Kettinger, and James Teng. Business process reengineering. *Information Systems Management*, 10:13–22, 1993.

[GLR+02] Anna Gerber, Michael Lawley, Kerry Raymond, Jim Steel, and Andrew Wood. Transformation: The missing link of mda. In *1st International Conference on Graph Transformation*, page 90105, 2002.

[GME11] GME. Gme metamodeling environment. http://w3.isis.vanderbilt.edu/Projects/gme/meta.html, 2011.

[Gom05] Hassan Gomaa. *Designing software product lines with UML - from use cases to pattern-based software architectures.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2005.

[Gri00] Martin L. Griss. Implementing product-line features with component reuse. In *6th International Conerence on Software Reuse*, page 137152, 2000.

[Gro03] Object Management Group. Mda guide version 1.0.1. http://www.omg.org/docs/omg/03-06-01.pdf, 2003.

[Gro04] Object Management Group. Meta object facility (mof) 2.0 core specification. http://www.omg.org/cgi-bin/apps/docptc/03-10-04.pdf, 2004.

[Gro05] Object Management Group. Mof qvt final adopted specification, 2005.

[Gro06a] Object Management Group. Bpmn 1.0: Omg final adopted specification. http://www.omg.org/bpmn/Documents/OMGFinalAdoptedBPMN1.0Spec.pdf, 2006.

[Gro06b] Object Management Group. Object constraint language, version 2.0. http://www.omg.org/spec/OCL/2.0/PDF, May 2006.

[Gro07] Object Management Group. Unified modeling language (uml), version 2.1.2, omg infrastructure specification. http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF, 2007.

[GS08] Hassan Gomaa and Michael Eonsuk Shin. Multiple-view modelling and meta-modelling of software product lines. *IET Software*, 2:94–122, 2008.

[Har91] James Harrington. *Business process improvement: The breakthrough strategy for total quality, productivity, and competitiveness.* McGraw-Hill, New York, 1991.

[HC03] Michael Hammer and James Champy. *Reengineering the Corporation: A Manifesto for Business Revolution.* Harper Business Books, New York, 2003.

[Her97] Jose Antonio Hernandez. *The SAP R/3 Handbook.* McGraw-Hill & Osborne Media, 1997.

[HMPO+08] Oystein Haugen, Birger Moller-Pedersen, Jon Oldevik, Goran K. Olsen, and Andreas Svendsen. Adding standardized variability to domain specific languages. In *Software Product Line Conference*, 2008.

[HP03] Gunter Halmans and Klaus Pohl. Communicating the variability of a software-product family to customers. In *Software and System Modeling*, pages 15–36, 2003.

[HP10] L. Hillah and L. Petrucci. Standardisation des réseaux de petri : etat de l'art et enjeux futurs. *Genie Logiciel*, 93:5–10, 2010.

[HR04] David Harel and Bernhard Rumpe. Meaningful modeling: Whats the semantics of "semantics"? *IEEE Computer*, 37:6472, 2004.

[HRG83] A. W. Holt, H. R. Ramsey, and J.D. Grimes. Coordination system technology as the basis for a programming environment. *Electrical Communication*, 77:307313, 1983.

[HWK+06] L. Hotz, K. Wolter, T. Krebs, S. Deelstra, M. Sinnema, J. Nijhuis, and J. MacGregor. *Configuration in Industrial Product Families, The ConIPF Methodology*. IOS Press, 2006.

[IBK11] Paul Istoan, Nicolas Biri, and Jacques Klein. Issues in model-driven behavioural product derivation. In *5th Workshop on Variability Modeling of Software-Intensive Systems*, pages 69–78, 2011.

[Ini11] Workflow Patterns Initiative. Workflow patterns initiative. http://www.workflowpatterns.com, 2011.

[INR10] INRIA. Feature diagram editor. http://movida.gforge.inria.fr/uploads/Demos/FeatureDi 2010.

[IR05] S.I. Irny and A.A. Rose. Designing a strategic information systems planning methodology for malaysian institutes of higher learning. *Issues in Information System*, 6:23–31, 2005.

[Jay94] Nimal Jayaratna. *Understanding and Evaluating Methodologies: NIMSAD, a Systematic Framework*. McGraw-Hill, Inc. New York, NY, USA, 1994.

[JB96] Stefan Jablonski and Christoph Bussler. *Workflow Management: Modeling Concepts, Architecture and Implementation*. International Thomson Computer Press, September 1996.

[Jen92] Kurt Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use, Vol. 1: Basic Concepts*. Springer-Verlag, Berlin, 1992.

[Jen94] Kurt Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use, Vol. 2: Analysis Methods*. Springer-Verlag, Berlin, 1994.

[JGJ97] I. Jacobson, M.L. Griss, and P. Jonsson. *Software reuse: architecture, process and organization for business success*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1997.

[JK06] Frederic Jouault and Ivan Kurtev. Transforming models with atl. In *Lecture Notes in Computer Science*, page 128138. Springer Berlin Heidelberg, 2006.

[JK09] Kurt Jensen and Lars M. Kristensen. *Coloured Petri Nets. Modelling and Validation of Concurrent Systems*. Springer-Verlag, Berlin Heidelberg, 2009.

[JKW07] Kurt Jensen, Lars Michael Kristensen, and Lisa Wells. Coloured petri nets and cpn tools for modelling and validation of concurrent systems. *Int. J. Softw. Tools Technol. Transf.*, 9:213–254, 2007.

[JTC07] Technical Committee ISO/IEC JTC1. Iso/iec 15909 : Software and system engineering highlevel petri nets. Technical report, ISO, 2007.

[KCH+90] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

[Ken02] Stuart Kent. Model driven engineering. In *Integrated Formal Methods*, pages 286–298, 2002.

[Ken03] Stuart Kent. Model driven language engineering. *Electronic Notes in Theoretical Computer Science*, 72:286–298, 2003.

[KKL+98a] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. Form: A feature-oriented reuse method with domain-specific reference architectures. *Ann. Softw. Eng.*, 5:143–168, 1998.

[KKL+98b] Kyo Chul Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. Form: A feature-oriented reuse method with domain-specific reference architectures. *Ann. Softw. Eng.*, 5:143–168, 1998.

[Kle08] Anneke Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Professional, 2008.

[KMHC05] Soo Dong Kim, Hyun Gi Min, Jin Sun Her, and Soo Ho Chang. Dream: A practical product line engineering using model driven architecture. In *ICITA*, page 7075, 2005.

[Kru06] Charles W. Krueger. New methods in software product line development. In *SPLC*, pages 95–102. IEEE Computer Society, 2006.

[KSN05] Chen Kai, Janos Sztipanovits, and Sandeep Neema. Toward a semantic anchoring infrastructure for domain-specific modeling languages. In *5th ACM International Conference On Embedded Software*, pages 18–22, 2005.

[KWB03] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained. The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.

[LCB96] R. Lakin, N. Capon, and N. Botten. Bpr enabling software for the financial services industry. *Management services*, 40:18–25, 1996.

[LDL03] Ann Lindsay, Denise Downs, and Ken Lunn. Business processes–attempts to find a definition. *Information and Software Technology*, 45:1015–1019, 2003.

[Lev60] Theodore Levitt. Marketing myopia. *Harvard Business Review*, 82:13849, 1960.

[LGB08] M. A. Laguna and B. Gonzalez-Baixauli. Product line requirements: Multi-paradigm variability models. In *11th Workshop on Requirements Engineering*, 2008.

[LMRC11] Miguel A. Laguna, Jose M. Marques, and Guillermo Rodriguez-Cano. Feature diagram formalization based on directed hypergraphs. *Comput. Sci. Inf. Syst.*, 8:611–633, 2011.

[LW99] Dean Leffingwell and Don Widrig. *Managing software requirements: a unified approach.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[MBB+84] R. Maddison, G. Baker, L. Bhabuta, G. Fitzgerald, K. Hindle, J. Song, N. Stokes, and J. Wood. Feature analysis of five information systems methodologies. In T. Bemelmans, editor, *Beyond Productivity: Information Systems Development For Organizational Effectiveness.* Elsevier Science Publishers, North Holland, 1984.

[MCVG05] Tom Mens, Krzysztof Czarnecki, and Pieter Van Gorp. A taxonomy of model transformations. In *Language Engineering for Model-Driven Software Development, in Dagstuhl Seminar Proceedings*, 2005.

[Men07] J. Mendling. *Detection and prediction of errors in epc business process models.* PhD thesis, Vienna University of Economics and Business Administration, Vienna, Austria, 2007.

[MH05] Alessandro Maccari and Anders Heie. Managing infinite variability in mobile terminal software. *Software: Practice and Experience*, 35:513 – 537, 2005.

[Mie09] D. Miers. Bpm: driving business performance. http://www.bptrends.com, 2009.

[MKKJ10] Brice Morin, Jacques Klein, Joerg Kienzle, and Jean-Marc Jezequel. Flexible model element introduction policies for aspect-oriented modeling. In *13th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, LNCS 6395, pages 63–77. Springer, 2010.

[Mos10] Sebastien Mosser. *Behavioral Compositions in Service Oriented Architecture.* PhD thesis, Polytech'Nice - Sophia Antipolis, October 2010.

[MP08] Ivan Markovic and Alessandro Costa Pereira. Towards a formal framework for reuse in business process modeling. In *Business Process Management Workshops*, pages 484–495, 2008.

[MPH+07] Andreas Metzger, Klaus Pohl, Patrick Heymans, Pierre-Yves Schobbens, and Germain Saval. Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In *Requirements Engineering*, pages 243–253, 2007.

[MPL+09] Brice Morin, Gilles Perrouin, Philippe Lahire, Gilles Barais, Olivuer Vanwormhoudt, and Jean-Marc Jezequel. Weaving variability into domain metamodels. In *MoDELS*, pages 690–705, 2009.

[MS03] Jason Xabier Mansell and David Sellier. Decision model and exible component de

nition based on xml technology. In *PFE*, 2003.

[Mur89] T. Murata. Petri nets: Properties, analysis and applications. *Proc. IEEE*, 77:541–580, 1989.

[MW12] Merriam-Webster. Merriam-webster online dictionary. http://www.merriam-webster.com/dictionary/methodology, 2012.

[MWH99] Florian Matthes, Holm Wegner, and Patrick Hupe. A process-oriented approach to software component definition. In *CAiSE*, pages 26–40, 1999.

[Nau63] Peter Naur. Revised report on the algorithmic language algol 60. *Commun. ACM*, 6:1–23, 1963.

[NN92] Hanne Riis Nielson and Flemming Nielson. *Semantics with applications: a formal introduction.* John Wiley and Sons, Inc., New York, NY, USA, 1992.

[Nor99] Linda Northrop. A framework for software product line practice. In *Proceedings of the Workshop on Object-Oriented Technology*, pages 365 – 376. Springer-Verlag London, 1999.

[OAS07] OASIS. Web services business process execution language (ws-bpel), version 2.0. oasis standard. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf, 2007.

[OMG01] OMG. Model driven architecture. http://www.omg.org/mda/, June 2001.

[OMG03] OMG. Mda guide version 1.0.1, June 2003.

[OMG06] OMG. Object constraint language. http://www.omg.org/spec/OCL/2.0/PDF, May 2006.

[OMG11] OMG. Business process model and notation (bpmn) version 2.0. http://www.omg.org/spec/BPMN/2.0/, 2011.

[Par76] David Lorge Parnas. On the design and development of program families. *Transactions on Software Engineering*, 2:1–9, 1976.

[PBvdL05] Klaus Pohl, Gunter Bockle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer, 2005.

[Per06] Gilles Perrouin. Coherent integration of variability mechanisms at the requirements elicitation and analysis levels. In *Workshop on Managing Variability for Software Product Lines*, 2006.

[Pet62] C.A. Petri. *Kommunikation mit Automaten.* PhD thesis, Technische University Darmstadt, 1962.

[Pet77] J.L. Peterson. Petri nets. *Computing Surveys*, 9:223252, 1977.

[Pet80] James L. Peterson. A note on coloured petri nets. *Information Processing Letters*, 11:40–43, 1980.

[PKGJ08a] Gilles Perrouin, Jacques Klein, Nicolas Guelfi, and Jean-Marc Jezequel. Reconciling automation and flexibility in product derivation. In *SPLC*, page 339348, 2008.

[PKGJ08b] Gilles Perrouin, Jacques Klein, Nicolas Guelfi, and Jean-Marc Jezequel. Reconciling automation and flexibility in product derivation. In *12th International Software Product Line Conference*, pages 339–348. IEEE Computer Society, Washington, DC, USA, 2008.

[Poh03] Risto Pohjonen. Boosting embedded systems development with domain-specific modeling. In *RTC Magazine*, page 5761. 2003.

[Pur] Pure::Variants. Puresystems. http://www.pure-systems.com/.

[RC10] Julia Rubin and Marsha Chechik. From products to product lines using model matching and refactoring. In *Workshop on Model-driven Approaches in Software Product Line Engineering*, 2010.

[Rec10] Jan C. Recker. Opportunities and constraints : the current struggle with bpmn. *Business Process Management Journal*, 16:181–201, 2010.

[Rie03] Matthias Riebisch. Towards a more precise definition of feature models. In *Modelling Variability for Object-Oriented Product Lines*, pages 64–76, 2003.

[RIRG05] Jan Recker, Marta Indulska, Michael Rosemann, and Peter Green. Do process modeling techniques get better? a comparative ontological analysis of bpmn. In *16th Australasian Conference on Information Systems*, 2005.

[RIRG06] Jan Recker, Marta Indulska, Michael Rosemann, and Peter Green. How good is bpmn really? insights from theory and practice. In *14th European Conference on Information Systems*, pages 1582–1593, 2006.

[RWL+03] Anne V. Ratzer, Lisa Wells, Henry Michael Lassen, Mads Laursen, Jacob Frank Qvortrup, Martin Stig Stissing, Michael Westergaard, Soren Christensen, and Kurt Jensen. Cpn tools for editing, simulating, and analysing coloured petri nets. In *Applications and Theory of Petri Nets*, volume 2679, pages 450–462, 2003.

[SAJK02] A.-W. Scheer, F. Abolhassan, W. Jost, and M. Kirchmer. *Business Process Excellence ARIS in Practice*. Springer-Verlang, Berlin, 2002.

[Sch00] A.W. Scheer. *ARIS: Business Process Modelling*. Springer-Verlag, Berlin, 2000.

[SDH06] Marco Sinnema, Sybren Deelstra, and Piter Hoekstra. The covamof derivation process. In *ICSR*, pages 101–114, 2006.

[SDNB04] Marco Sinnema, Sybren Deelstra, Jos Nijhuis, and Jan Bosch. Covamof: A framework for modeling variability in software product families. In *SPLC*, pages 197–213, 2004.

[SEI] SEI. Software product line conference - hall of fame. http://splc.net/fame.html.

[Sei03] E. Seidewitz. What models mean. *IEEE Software*, 20:2632, 2003.

[SEI12] SEI. Framework for software product line practice, version 5.0. http://www.sei.cmu.edu/productlines/tools/framework/, 2012.

[Sel03] Bran Selic. The pragmatics of model-driven development. *IEEE Software*, 20:1925, 2003.

[Sel06] Bran Selic. Model-driven development: Its essence and opportunities. In *International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, page 313319. IEEE Computer Society Press, 2006.

[SF03] Howard Smith and Peter Fingar. *Business Process Management: The Third Wave*. Meghan-Kiffer Press, 2003.

[SGS+04] Greg Straw, Geri Georg, Eunjee Song, Sudipto Ghosh, Robert France, and James Bieman. Model composition directives. In *7th International Conference Unified Modelling Language: Modelling Languages and Applications*, 2004.

[SHTB07] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51:456–479, 2007.

[SJ04] Klaus Schmid and Isabel John. A customizable approach to full lifecycle variability management. *Sci. Comput. Program*, 53:259–284, 2004.

[SK95] Kenneth Slonneger and Barry L. Kurtz. *Formal syntax and semantics of programming languages - a laboratory based approach*. Addison-Wesley, Boston, MA, USA, 1995.

[SK02] Shane Sendall and Wojtek Kozaczynski. Model transformation the heart and soul of model-driven software development. Technical report, Ecole Polytechnique Federale de Lausanne, 2002.

[SL05] Kamyar Sarshar and Peter Loos. Comparing the control-flow of epc and petri net from the end-user perspective. In *International Conference on Business Process Management*, page 434439, 2005.

[SPL] SPL. Welcome to software product lines. http://www.softwareproductlines.com/.

[SvGB05] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. A taxonomy of variability realization techniques. *Software Practice and Experience*, 35:705–754, 2005.

[Tae04] Gabriele Taentzer. Agg: A graph transformation environment for modeling and validation of software. In *Workshop on Applications of Graph Transformations with Industrial Relevance*, 2004.

[Tes84] Lawrence G. Tesler. Programming languages. *Scientific American*, 251:70–78, 1984.

[Tra05] Laurence Tratt. Model transformations and tool integration. *Journal of Software and Systems Modeling*, 4:112–122, 2005.

[vdA98] Wil M. P. van der Aalst. The application of petri nets to workflow management. *Journal of Circuits, Systems and Computers*, 8:21–66, 1998.

[vdA00] Wil M. P. van der Aalst. Workflow verification: Finding control-flow errors using petri-net-based techniques. In *Business Process Management*, pages 161–183, 2000.

[vdA03] Wil M. P. van der Aalst. Challenges in business process management: Verification of business processing using petri nets. *Bulletin of the EATCS*, 80:174–199, 2003.

[vdAtH99] Wil van der Aalst and Arthur ter Hofstede. Workflow patterns. http://www.workflowpatterns.com/, 1999.

[vdAtH05] W. M. P. van der Aalst and A. H. M. ter Hofstede. Yawl: Yet another workflow language. *Information Systems*, 30:245275, 2005.

[vdAtHW03] W.M.P. van der Aalst, A.H.M. ter Hofstede, and M. Weske. Business process management: a survey. In *Proceedings of the 2003 international conference on Business process management*, pages 1–12. Springer-Verlag, 2003.

[vdAvHtH+11] W.M.P. van der Aalst, K.M. van Hee, A.H.M. ter Hofstede, N. Sidorova, H.M.W. Verbeek, M. Voorhoeve, and M.T. Wynn. Soundness of workflow nets: classification, decidability, and analysis. *Formal Aspects of Computing*, 23:333–363, 2011.

[VDKV00] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35:26–36, 2000.

[vdL02] F. van der Linden. Software product families in europe: The esaps and cafe projects. *IEEE Software*, 19:4149, 2002.

[VG07] Markus Voelter and Iris Groher. Product line implementation using aspect-oriented and model-driven software development. In *11th International Software Product Line Conference*, pages 233–242. IEEE Computer Society Washington, DC, USA, 2007.

[vGBS01] Jilles van Gurp, Jan Bosch, and Mikael Svahnberg. On the notion of variability in software product lines. In *WICSA*, pages 45–54, 2001.

[vO02] Rob van Ommering. Building product populations with software components. In *24th International Conference on Software Engineering*, page 255265, 2002.

[VVP02] Daniel Varro, Gergely Varro, and Andras Pataricza. Designing the automatic transformation of visual languages. *Science of Computer Programming*, 44:205227, 2002.

[WCR09] Nathan Weston, Ruzanna Chitchyan, and Awais Rashid. A framework for constructing semantically composable feature models from natural language requirements. In *13th International Software Product Line Conference*, pages 211–220, 2009.

[Wes07] M. Weske. *Business Process Management: Concepts, Languages, Architectures*. Springer-Verlag New York Inc., 2007.

[WGHS99] Mathias Weske, Thomas Goesmann, Roland Holten, and Rudiger Striemer. A reference model for workflow application development process. In *International Joint Conference on Work Activities Coordination and Collaboration*, pages 1–10, 1999.

[Whi04] S. A. White. Introduction to bpmn. Technical report, IBM Cooperation, 2004.

[Wik] Wikipedia. Feature model. http://en.wikipedia.org/wiki/FeatureModel.

[Wit96] James Withey. Investment analysis of software assets for product lines. Technical Report CMU/SEI-96-TR-010, Software Engineering Institute, Carnegie Mellon University, 1996.

[WK03a] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2003.

[WK03b] Michael Weber and Ekkart Kindler. Petri net technology for communication-based systems. *Advances in Petri Nets*, 2472:124–144, 2003.

[WL99] D. M. Weiss and R. Lai. *Software Product-Line Engineering: a Family-Based Software Development Process*. Addison-Wesley, Reading, Massachusetts, 1999.

[Wor12] Petri Nets World. The petri nets bibliography. http://www.informatik.uni-hamburg.de/TGI/PetriNets/bibliographies/aboutpnbibl.html, 2012.

[WS05] Terje Wahl and Guttorm Sindre. An analytical evaluation of bpmn using a semiotic quality framework. In K. Siau T. Halpin and J. Krogstie, editors, *Workshop on EvaluatingModeling Methods for Systems Analysis and Design*, page 533544. Eds., FEUP, Porto, Portugal, 2005.

[Xac] Xactium. Xmf-mosaic. http://www.xactium.com/.

[Xte07] Xtext. Xtext. http://www.eclipse.org/Xtext/, 2007.

[Zai97] Mohamed Zairi. Business process management: a boundaryless approach to modern competitiveness. *Business Process Management Journal*, 3:64–80, 1997.

[ZHJ03] Tewfik Ziadi, Loic Helouet, and Jean-Marc Jezequl. Towards a uml profile for software product lines. In *PFE*, pages 129–139, 2003.

[ZJ06a] Tewfik Ziadi and Jean-Marc Jezequel. Software product line engineering with the uml: deriving products. *Software Product Lines*, 1:557 – 588, 2006.

[ZJ06b] Tewfik Ziadi and ean-Marc JJezequel. Software product line engineering with the uml: Deriving products. In *Software Product Lines*, pages 557–588. Springer, 2006.

[ZM04] M. Zur Muehlen. *Workflow-based Process Controlling. Foundation, Design, and Application of workflow-driven Process Information Systems.* Logos Berlin, 2004.

[zMR08] Michael zur Muehlen and Jan C. Recker. How much language is enough? theoretical and practical use of the business process modeling notation. *CAiSE, Lecture Notes in Computer Science*, 5074:465–479, 2008.

APPENDIX

A. ANNEX 2: BUSINESS PROCESS FRAGMENTS FOR THE BCMS CASE STUDY

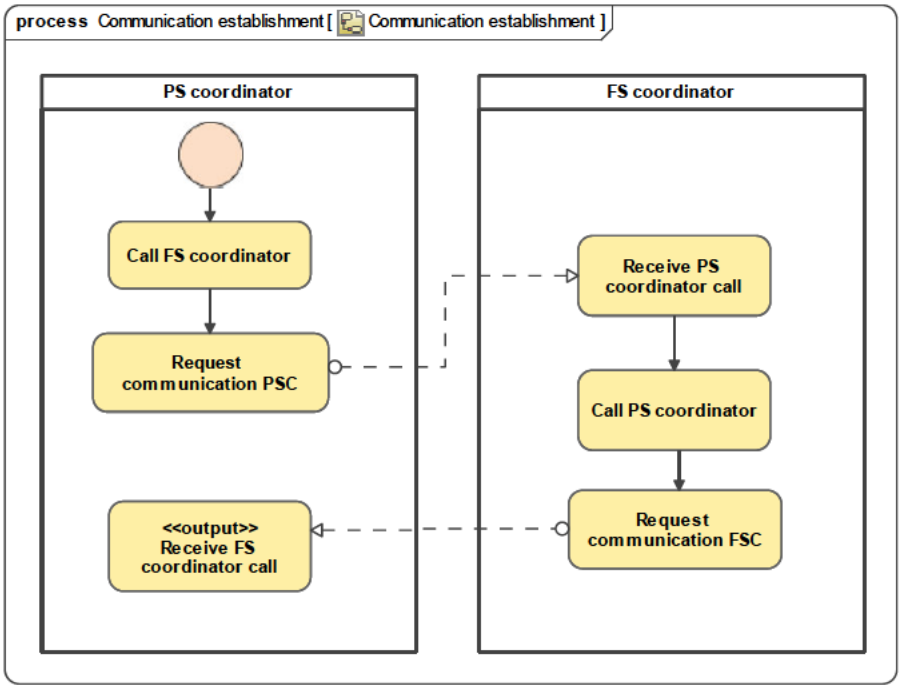


Fig. A.1: Communication establishment business process fragment

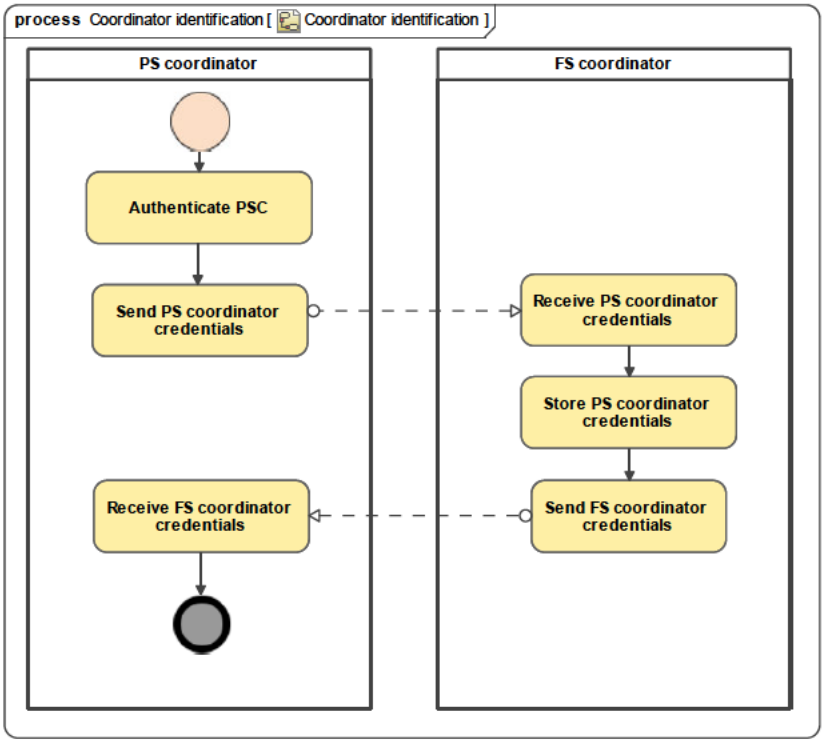


Fig. A.2: Coordinator identification business process fragment

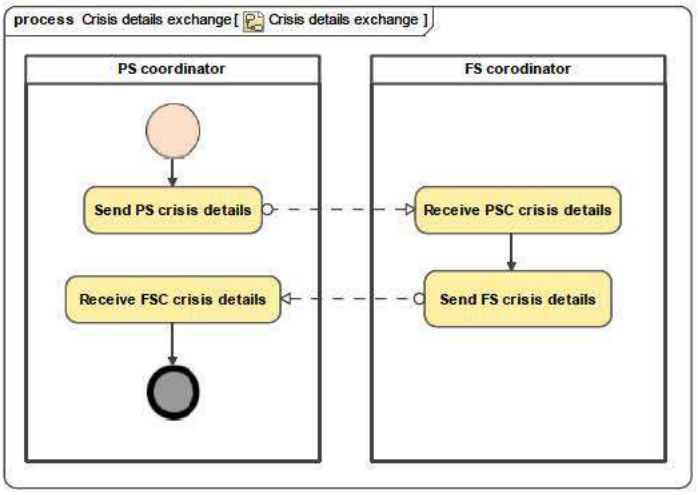


Fig. A.3: Crisis details exchange business process fragment

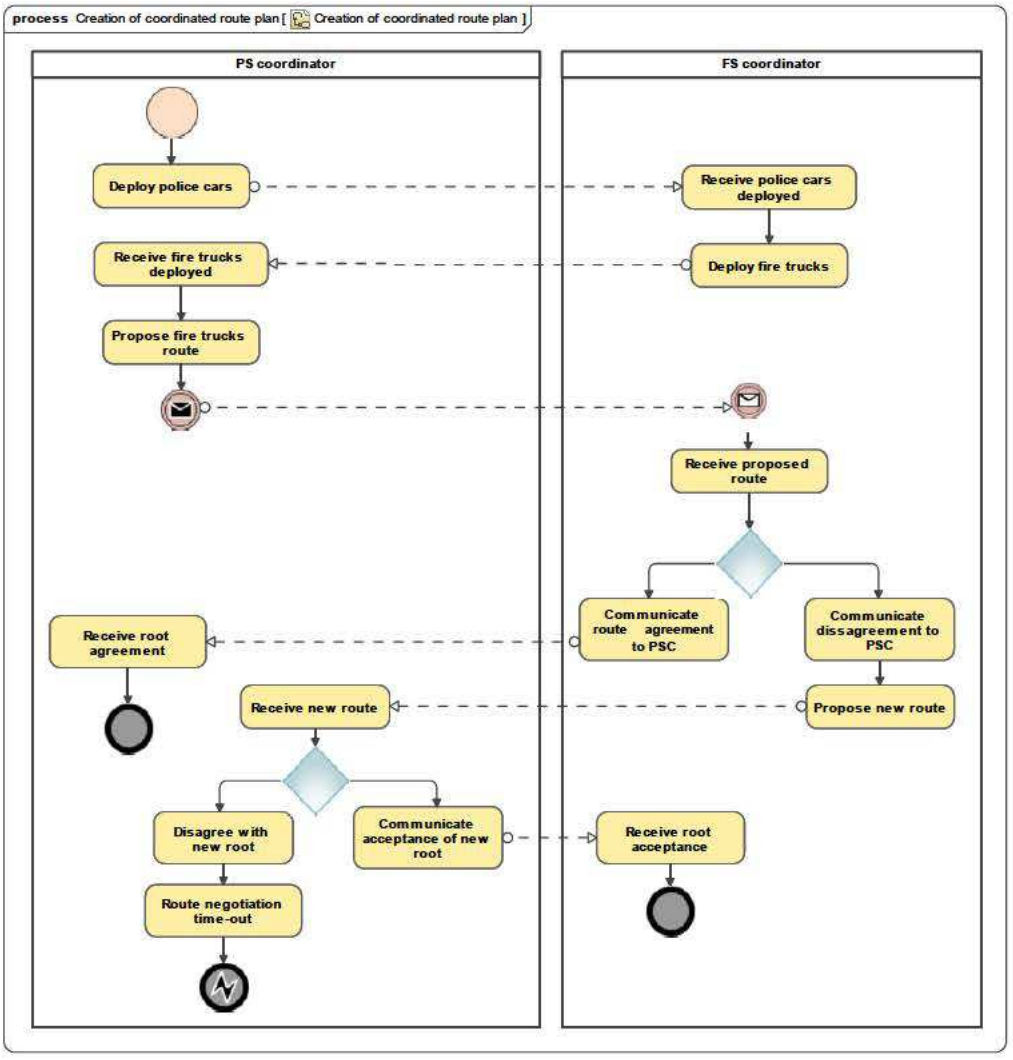


Fig. A.4: Creation of coordinated route plan business process fragment

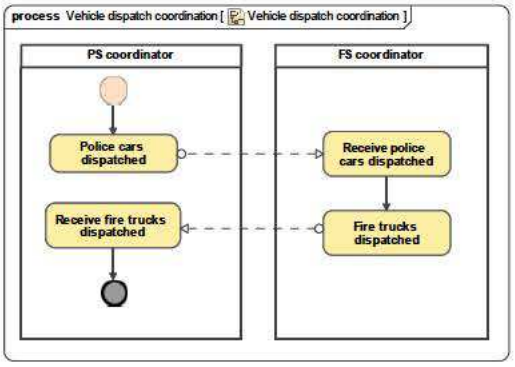


Fig. A.5: Vehicle dispatch coordination business process fragment

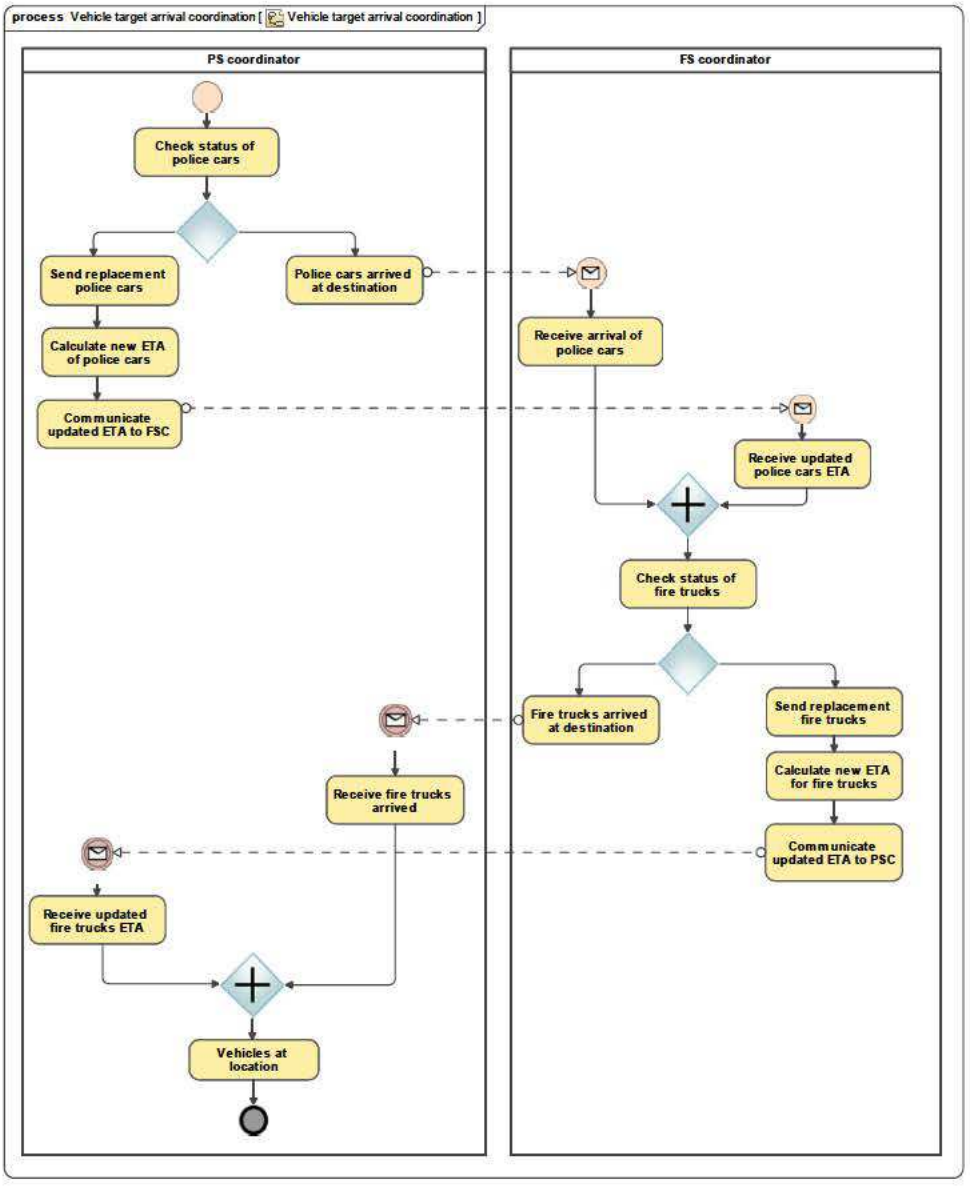


Fig. A.6: Vehicle target arrival coordination business process fragment

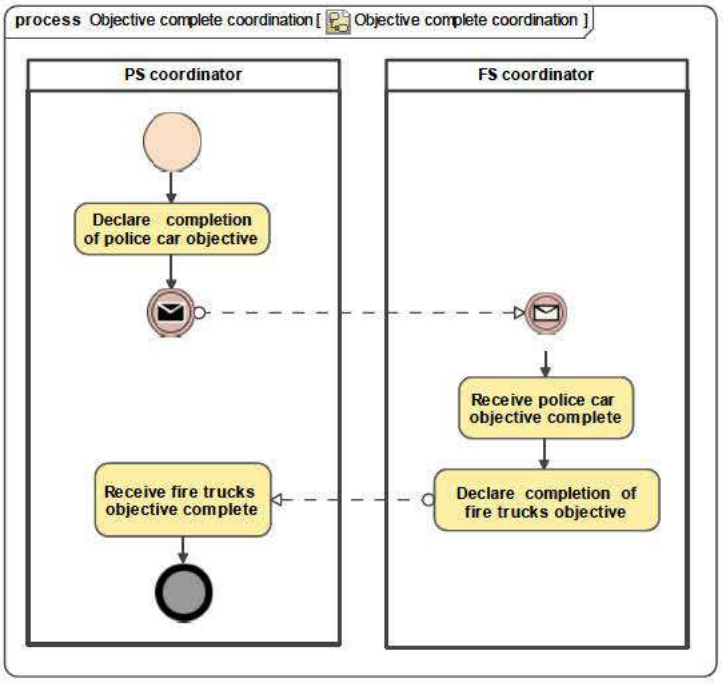


Fig. A.7: Crisis objective complete business process fragment

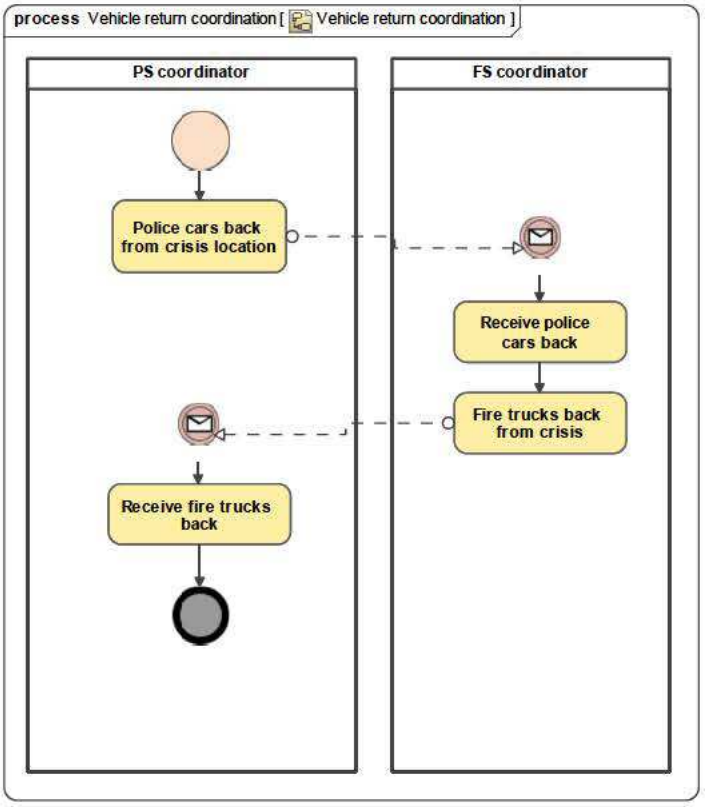


Fig. A.8: Vehicle return coordination business process fragment

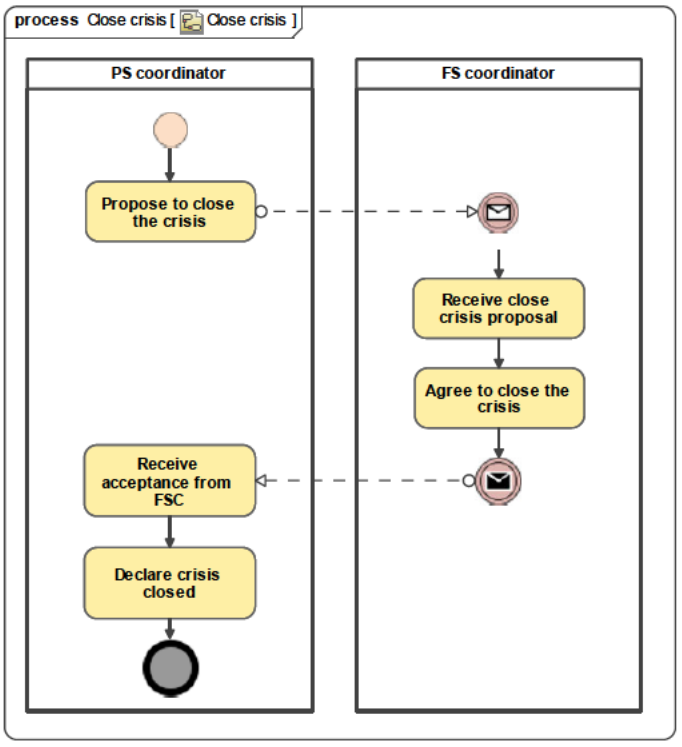


Fig. A.9: Close crisis business process fragment

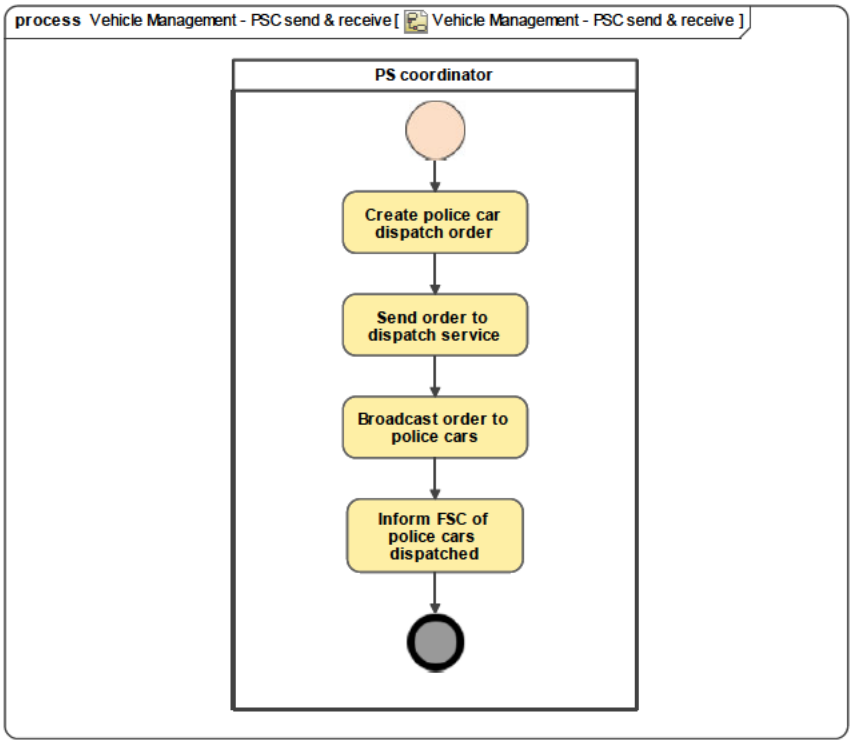


Fig. A.10: Vehicle management - PSC send and receive business process fragment

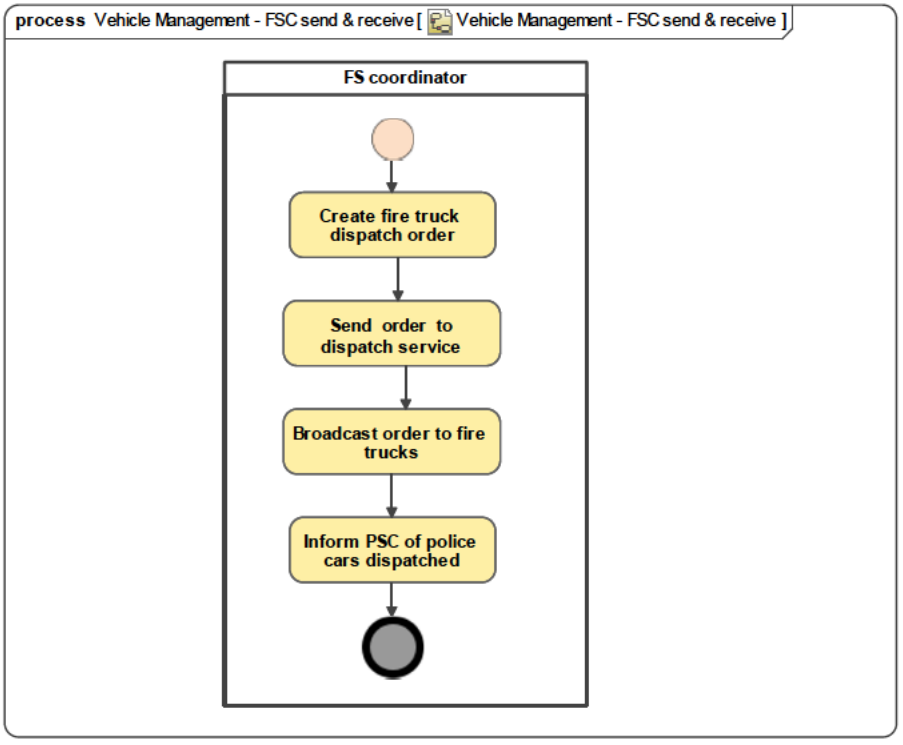


Fig. A.11: Vehicle management - FSC send and receive business process fragment

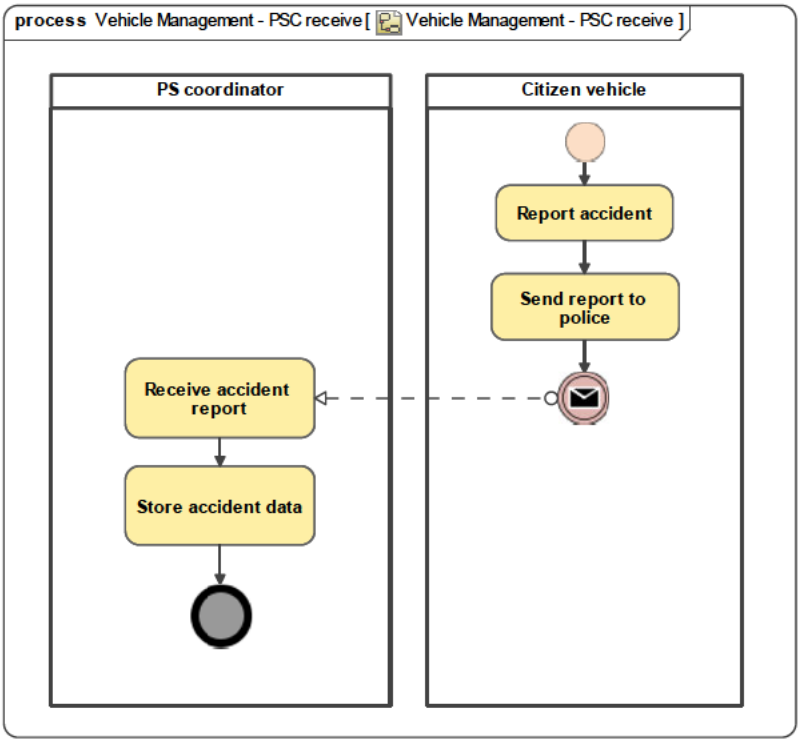


Fig. A.12: Vehicle management - PSC receive business process fragment

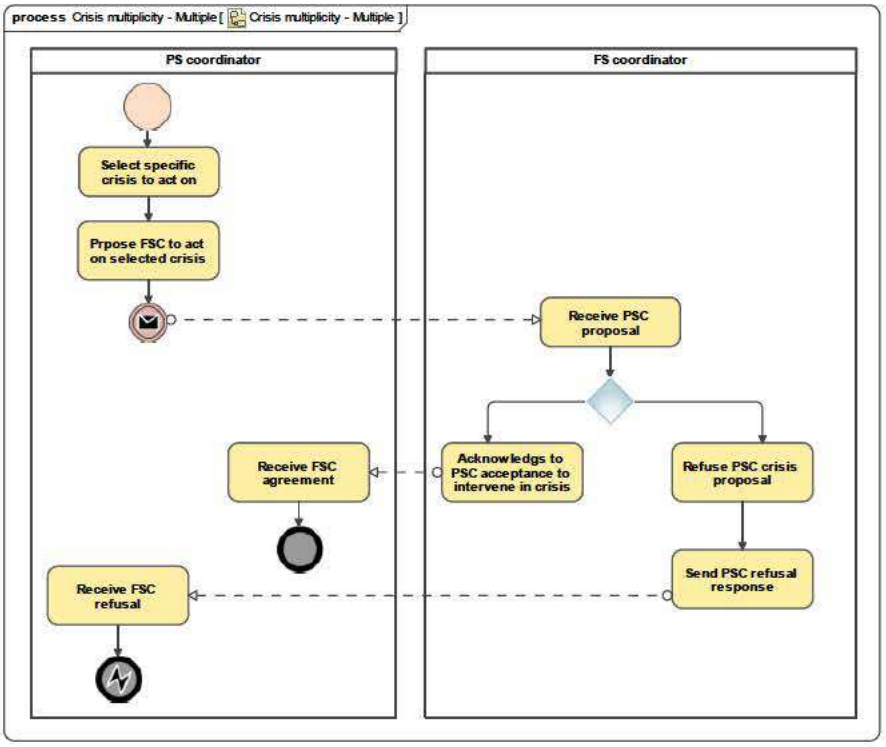


Fig. A.13: Multiple crisis business process fragment

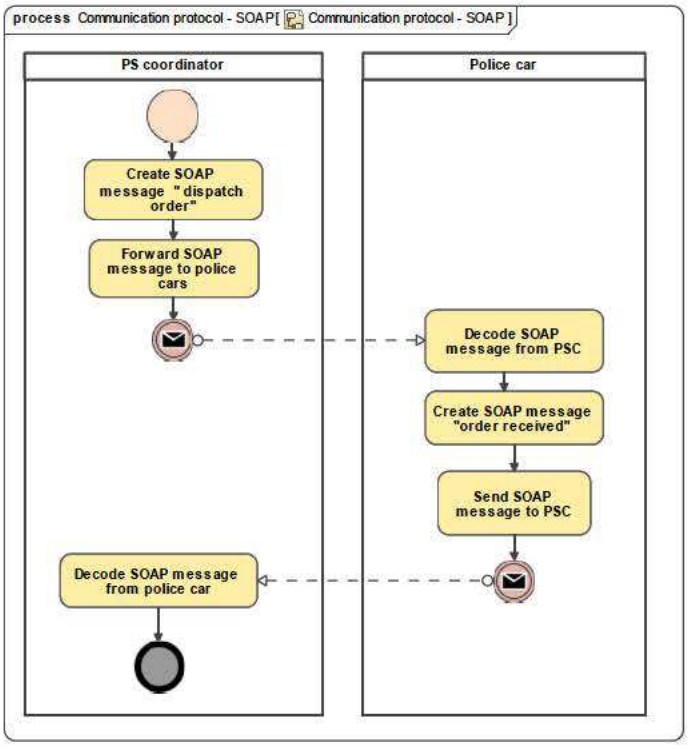


Fig. A.14: SOAP communication protocol business process fragment

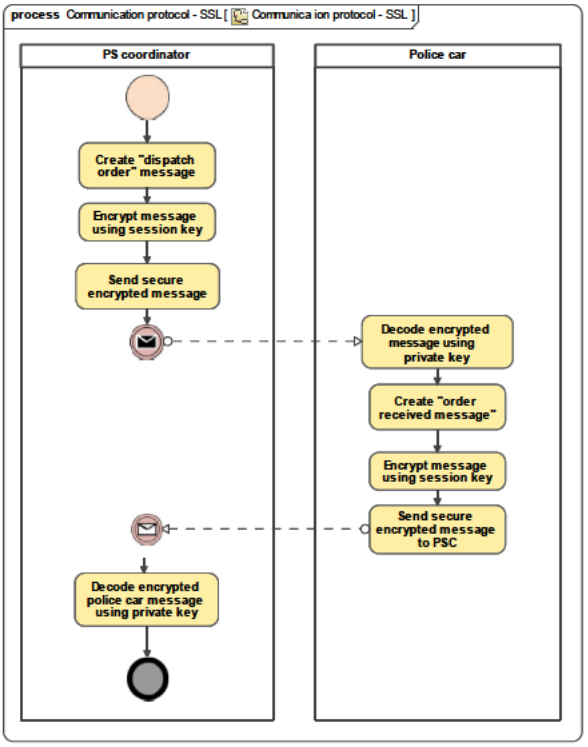


Fig. A.15: SSL communication protocol business process fragment

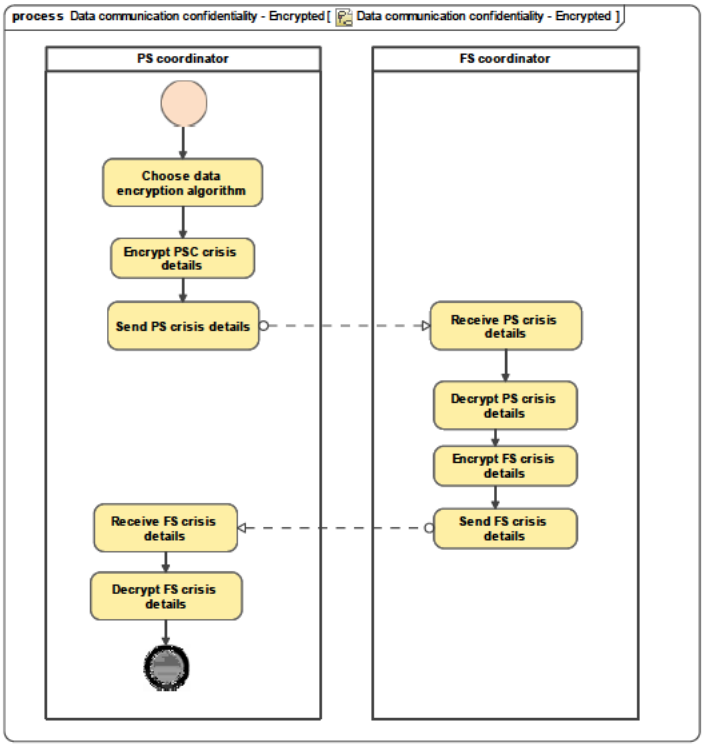


Fig. A.16: Encrypted data communication business process fragment

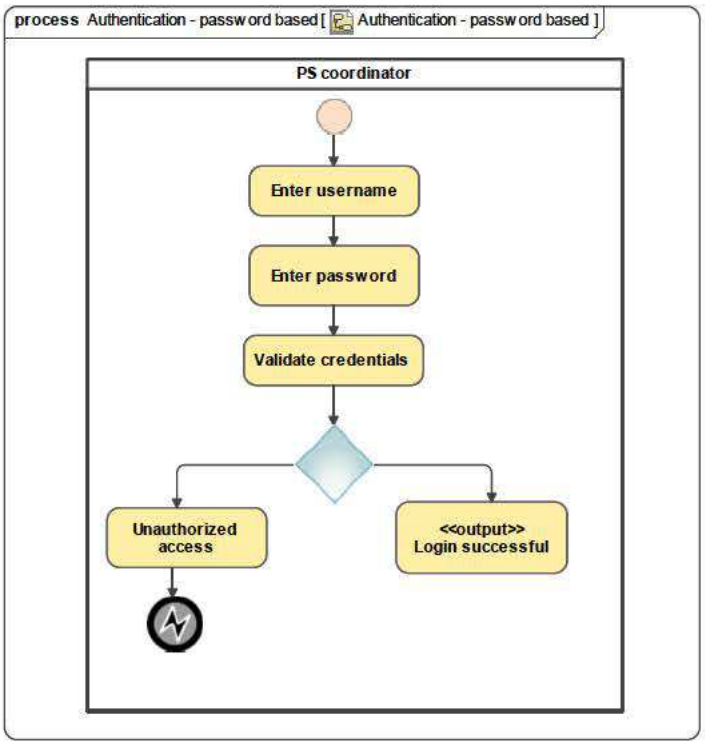


Fig. A.17: Password based authentication business process fragment

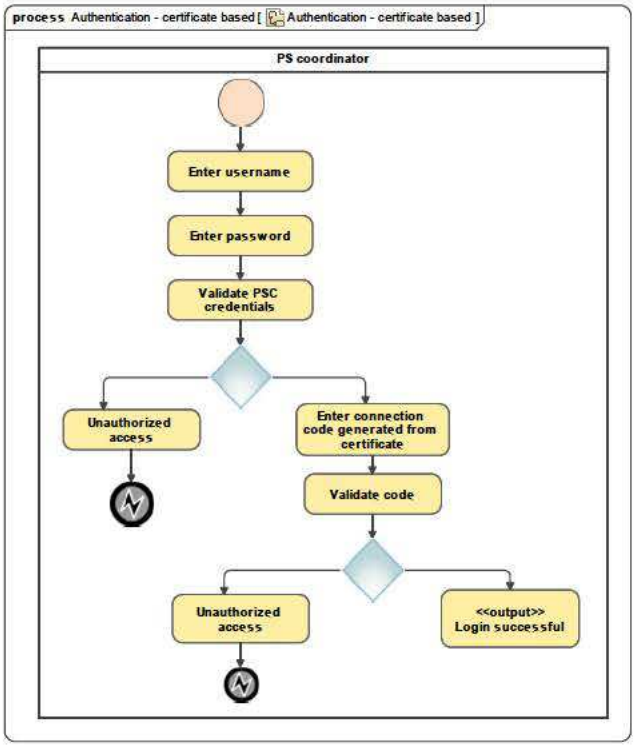


Fig. A.18: Certificate based authentication business process fragment

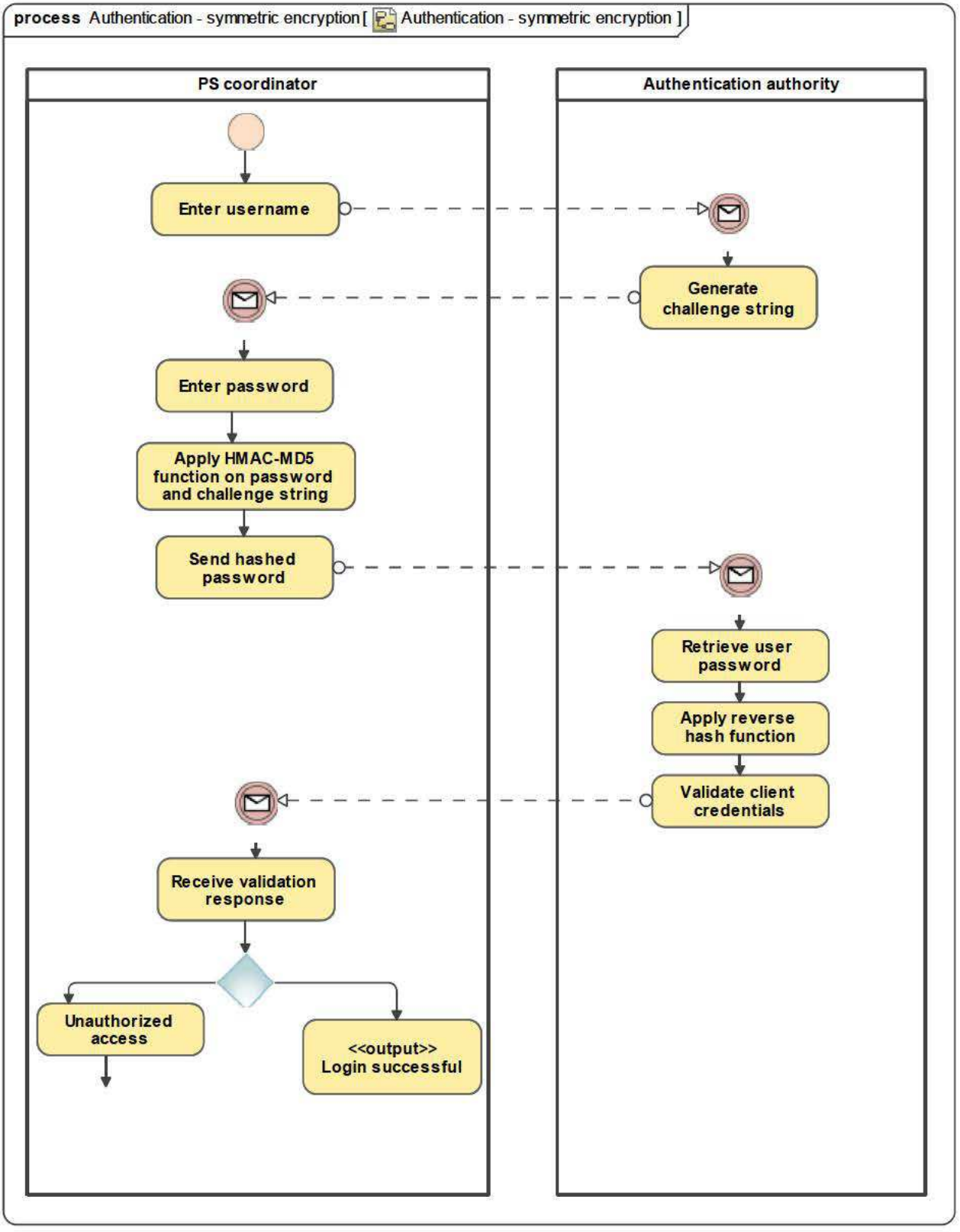


Fig. A.19: Symmetric encryption authentication business process fragment

Academic Use Only

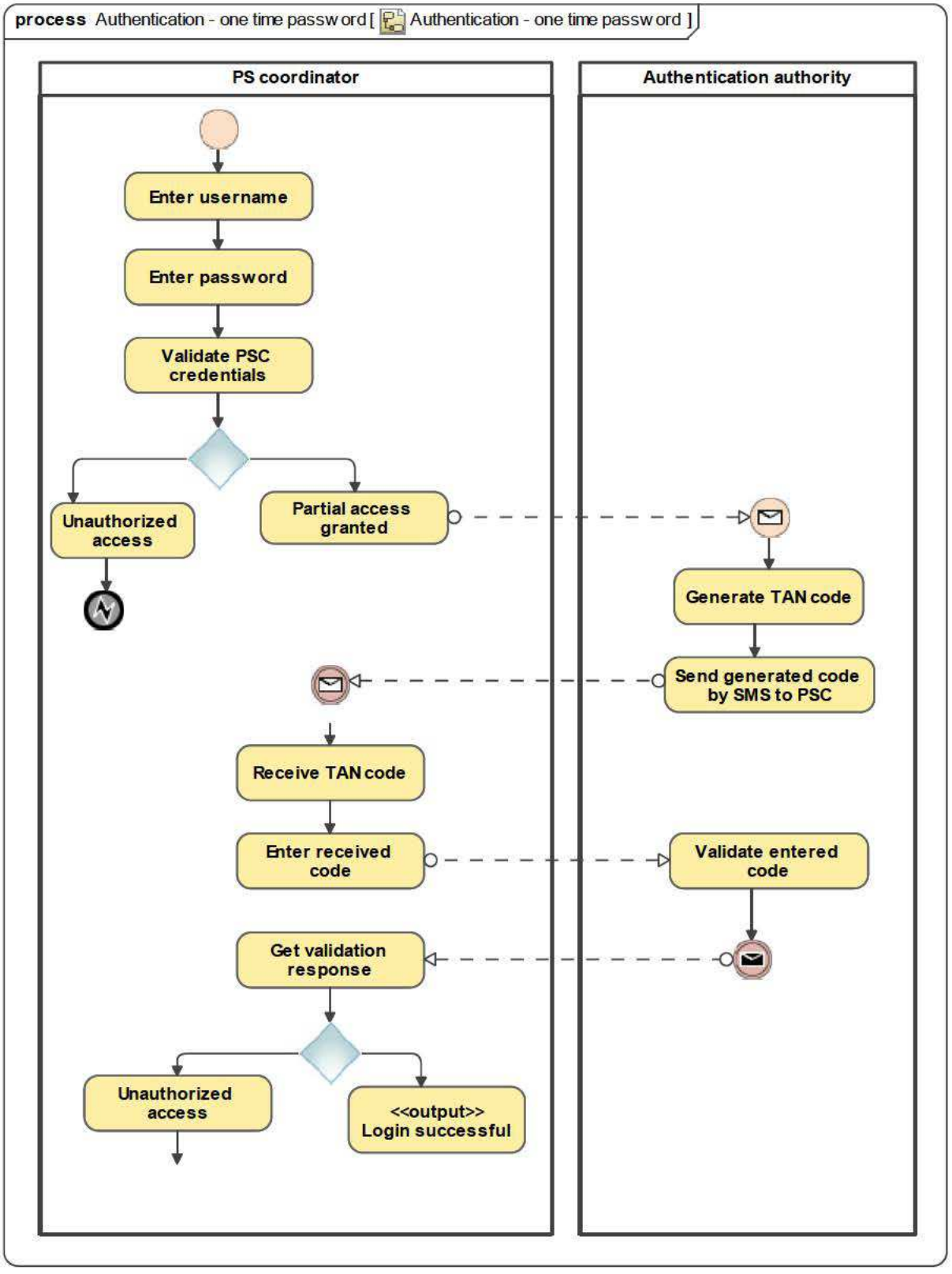


Fig. A.20: One time password based authentication business process fragment

Academic Use Only

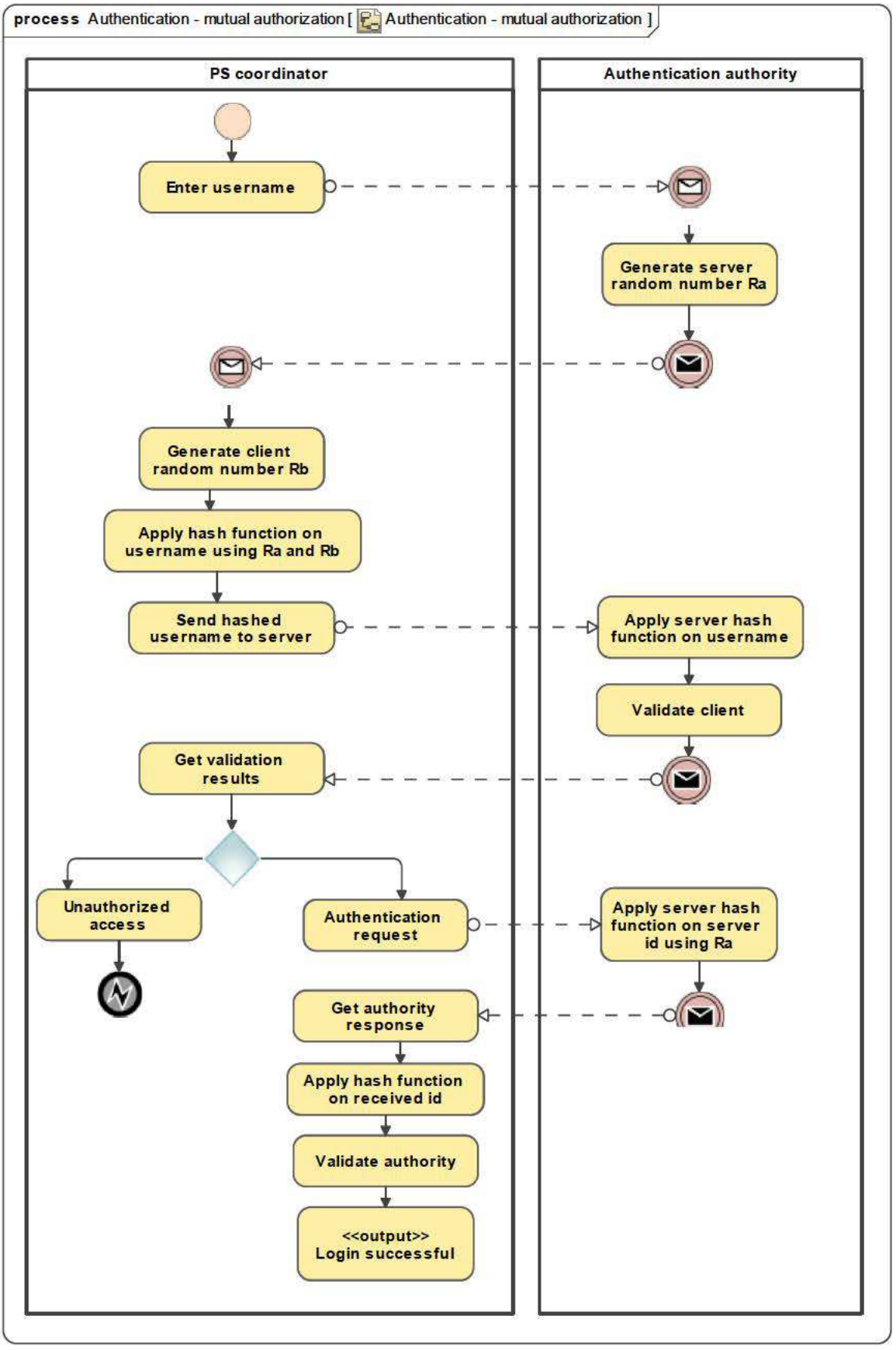


Fig. A.21: Authentication based on mutual authorization business process fragment

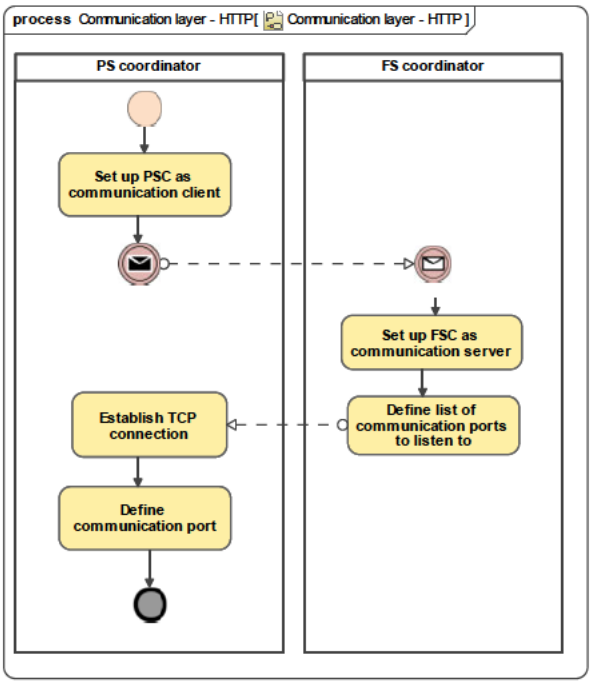


Fig. A.22: HTTP based communication layer business process fragment

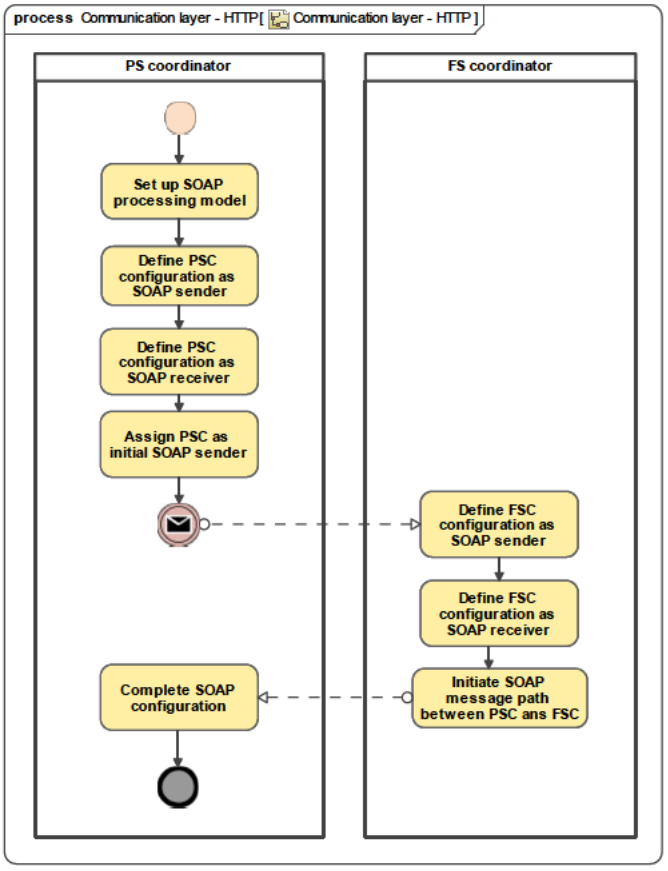


Fig. A.23: SOAP based communication layer business process fragment