# On the adoption of the business process execution language in the Everst Knowledge Framework

Niks, R.W.C.

*Award date:*
2008

**EINDHOVEN UNIVERSITY OF TECHNOLOGY**
**Department of Mathematics and Computer Science**

# On the adoption of the Business Process Execution Language in the Everest Knowledge Framework

by

R.W.C. Niks

Supervisors:

prof. dr. ir. G.J.P.M. Houben (TU/e)
ir. ing. M. Mastop (Everest)
drs. L. Hermans (Everest)

Eindhoven, June 2008

*“Faith consists in believing when it is beyond the power of reason to believe.”*

Voltaire (November 21, 1694 - May 30, 1778)

EINDHOVEN UNIVERSITY OF TECHNOLOGY

# *Abstract*

Business Information Systems
Department of Mathematics and Computer Science

Master of Science

by R.W.C. Niks

The subject covered by this research project involves increasing the interoperability of the Everest Knowledge Framework by the adoption of the Business Process Execution Language. Everest aims at increasing the compliance and interoperability of their process implementation in the Everest Knowledge Framework and allow business engineers to construct models from which a more generic process implementation can be derived. The enterprise modeling paradigm will be used to make an assessment of the modeling and deployment strategy of Everest. Typical approaches adopted in the application development life cycle of Everest are: enterprise architecture and model driven architecture.

The first accomplishment of this thesis is to formalize the syntax of the languages used by Everest to model and implement their processes. Closing the gap between the Everest modeling language and the Business Process Execution Language requires and assessment of the domain and technical spaces of both languages. This assessment should reveal the issues and challenges which must be considered when transforming one language into another. Finally, we propose an approach to transform the Everest specific process models into code based on the Business Process Execution Language. Applying the transformation approach to a process example from the Everest practice, allows us to derive conclusions about the correctness and completeness of the transformation results.

For Everest the adoption of Business Process Execution Language implies the integration of a corresponding engine in the Everest Knowledge Framework. Therefore best practices will be proposed to discus the design decisions Everest should concider when adopting BPEL in the Everest Knowledge Framework.

# *Acknowledgements*

This Master Thesis is the result of a research project carried out from January 2007 to June 2008 at the research group Information Systems at the Department of Mathematics and Computer Science at Eindhoven University of Technology. The research has been conducted in cooperation with Everest B.V. a company specialized in Business Knowledge Modeling & Engineering. For me, this thesis is the final step to obtain the Master of Science degree in Business Information Systems.

I would like to thank Geert-Jan Houben for his ideas and support during the project. As my supervisors at Everest, I would like to thank Leo Hermans and Mark Mastop for their patience and giving me the opportunity to conduct my research in a business setting. Also, I would like to thank Eric Verbeek, Edwin Popkin and Elco Rombouts for their willingness to provide me with valuable feedback.

Also Bram Hoefnagel, Bart van der Heijden, Koen Schuurmans and Jasper Verhoeven, for reviewing my work, your comments made this thesis certainly better reading material.

Finally, I would like to thank my relatives and girlfriend for the moral support, especially during the difficult periods of the project. And of course my friends and roommates, allowing me, when needed, to escape the reality of graduation.

It would never have been finished without you!

Eindhoven, June 2008

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

The rapid and constant changes that are very common in today's business environments affect not only business itself, but also its supporting Business Information Systems (BIS). Due to the dynamic market, the business needs to adapt to changes from its environment to remain competitive in that market. As a result, the business processes, business rules and business domain require continuous changes, renovation and adaptation to meet actual business needs. This and the increasing need for business agility will drive the rise for a new generation of BIS, which has implemented the business knowledge and process flow as business content. In this way the business content is interpreted by generic business engines responsible to derive the desired behavior directly from business content.

Three levels of business knowledge can be identified (see Figure 1.1): strategic, tactic and operational. Strategic business knowledge is represented by the business vision, the mission, goals and strategy. The goal-oriented business knowledge can be redefined to a point where it can be translated into tactical business knowledge expressed by: policies, rules, objectives, tactics, products and procedures that together will achieve the stated business goals [Rosca et al., 1995]. The tactic business knowledge is more or less represented by intentional rules and procedures that need to be expressed more specific such that they can be applied on the operational business level. The rules and procedures that end up in the implementation of an application are called 'executable rules' and 'executable processes', which refers to the rules and procedures that can be expressed formally, such that they can be interpreted and executed by computers. The goal is to automate the rules and procedures from the operative level to the system level, so that the transformation of rules and procedures from the operational business to its implementation is more transparent, by making the business rules and business processes explicit and independent of the 'programming logic' (software code). This

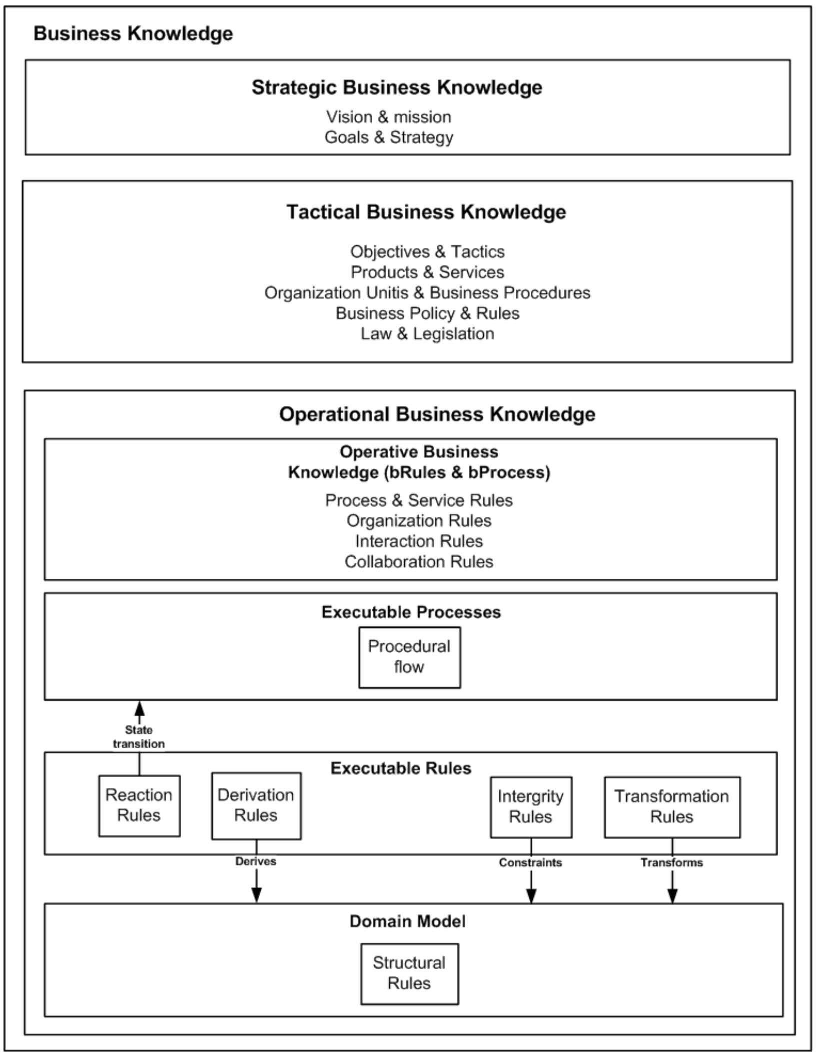


FIGURE 1.1: Levels of Business Knowledge.

implies that the rules constrains, derive data and transforms data in domain model and enables behavioral aspects (state transition) in the process.

Over the years, the business has sought ways to optimize the performance of business applications and increase the flexibility, when adapting to changes in the business environment. Automating the business implies that more and more operative rules and procedures must be translated into executable rules and executable processes, so that they can be deployed and enacted by 'business engines'. Following this approach the operative procedures are managed by the 'the world of business' and the executable processes by the 'the world of technology' [Morgan, 2002]. Enterprise modeling has been adopted by the industry as a widespread strategy for developing flexible enterprise applications in an evolutionary development life cycle. Both the enterprise architecture and Model Driven Architecture (MDA) have emerged from the enterprise modeling paradigm. The enterprise architecture focuses on the functional decomposition and integration of the business applications and MDA aims at providing high-level modeling language (supported by tools), such that models composed with these languages can be transformed into executable models, which can be enacted by business engines or implemented as programming logic.

## 1.1 Research Context

Everest is a company that is specialized in knowledge engineering, which is the process of acquisition, analyzing, validating and automating operational business knowledge from several domains (e.g.: mortgage, insurance and banking). To serve their customers, Everest has developed a knowledge based application development environment called the Everest Knowledge Framework (EKF). The EKF is used for the development of business applications, where business knowledge plays a central role when automating the decision making. By increasing the interoperability of the EKF with other systems and tools Everest aims at increasing their market share and maintaining their innovative position.

The ultimate goal of the EKF is to provide an application development environment, that is configurable and manageable by 'business-oriented people'. This requires that the supported tools and adopted modeling techniques must be understandable by the 'business-oriented people' [Hermans, 2007]. Although the development languages of the EKF are much less technical compared to traditional programming languages, it still requires people with engineering skills, qualified as 'business engineers', for modeling. Business-oriented people do usually not express operative business knowledge by means of models, but use natural language instead. For Everest the business engineers play an important role when survey the business demands and creating models toward a business solution. Because the business engineers have a more clear understanding of the business needs, they are better able to make decisions with respect to the business solution. Issues arise when the business engineers and 'software engineers' must synchronize the design and the implementation. Technical considerations could result in flaws or limitations of the application, such that frequent changes require increasing effort of communication between the business engineers and software engineers [Kontonya and Sommerville, 2002]. Therefore Everest has adopted an approach where higher-level models are developed by business engineers which are directly deployed onto the business engines. Following this approach Everest aims at an increase in agility and decrease of time-to-market, when developing business applications.

The customers of Everest have a demand for compliance in the business processes. This has resulted in the increasing interest of Everest to adopt a more generic approach to implement and deploy the processes in the EKF. Standardization in the business process domain is maturing and various languages and vocabularies have emerged from both the scientific and industry communities. Everest is mainly interested in the adoption of the Business Process Execution Language (BPEL) in the EKF, as it is at the time of writing the defacto standard for process enactment and interchange.

## 1.2 Problem Definition

The EKF supports modeling of certain aspects of applications in terms of executable models, which are enacted by business engines. Each model corresponds to a certain aspect of the system (e.g. domain, rule, process, task, etc.). Both the domain and rules play an increasingly important role in the EKF, as a consequence of the increasing degree to which primary customer-oriented processes from sale to delivery must be automated for different products via different channels. Everest uses a high-level modeling language, called the ‘EKF process language’, for modeling their business processes at the conceptual level. These conceptual models are implemented in terms of the EKF domain and rule models for which business engines are responsible to produce the desired process behavior. More specific, the domain and rule model are used as a basis for the implementation of the processes. The drawback of this approach is that the deployment of the EKF processes is Everest specific and lacks for interoperability with other applications and systems and therefore requires increasingly efforts of programming.

The EKF is used in various enterprise solutions where process interchanges becomes an increasingly important factor. Everest is interested in the adoption of BPEL as a language for process interchange or even integrate a BPEL engine within the EKF. In this thesis we therefore directly address the need of Everest to be able to translate their EKF process models into BPEL code. The transformation of one model into another, requires an assessment of the way Everest models and deploys its processes. For Everest this assessment is considered to be an important result, as no description of the syntax and semantics of the EKF process language was available at the start of this project. The insights into the syntax and semantics of the EKF process language should allow an evaluation of the contextual and conceptual differences between EKF process language and BPEL. The contextual and conceptual differences must be considered when bridging the gap between the EKF process language and BPEL, toward an approach to translate EKF process models into BPEL code. The assessment of the EKF process language is an important result for Everest, which raises questions about expressiveness of the

EKF process language compared to BPEL. For Everest the adoption of BPEL implies the integration of a BPEL engine in the EKF. Assessment of the consequences of the EKF process language and integration of a BPEL engine in the EKF, should result in a better understanding of the best practices which must be considered by Everest when adopting a BPEL engine in the EKF.

## 1.3 Research Questions

Several research questions arise from the objective as defined in the previous section. First, we specialize the objectives for modeling and deployment into the question: *'How the modeling and deployment is done from the enterprise modeling perspective?'*. Second, we determine: *'How the modeling and deployment of models is specifically done in the EKF'*. This requires an assessment of the syntax and semantics of the EKF process language, as no such specification was available at the start of this project. Third, we perform a similar assessment for BPEL: *'What are the key features and syntax of BPEL?'*. Fourth, we aim at answer the question: *'How to make the EKF more interoperable by adopting BPEL?'* . Conceivable, this question is divided into the following two partial questions: *'What are the issues and challenges when closing the gap between the EKF process language and BPEL?'* and *'How EKF process models can be transformed into BPEL code?'*. Finally, we reflect our solutions and discuss the limitations of our proposed solution, for which we aim at answering the question: *What are the best practices which must be considered when adopting our proposed solution?*. The following research questions should cover these objectives:

1. What are the principles used for enterprise modeling?
2. How does Everest apply enterprise modeling in the EKF?
3. What are the key features and syntax of BPEL?
4. How to make the EKF more interoperable by adopting BPEL?
5. What are the best practices which must be considered when adopting our proposed solution?

## 1.4 Thesis outline

For the outline of this thesis we distinguish the structure of: problem definition, analysis, design and evaluation (see Figure 1.2). As the problem definition and research objectives have already been discussed in the previous sections, we will now focus on the analysis, design and evaluation.

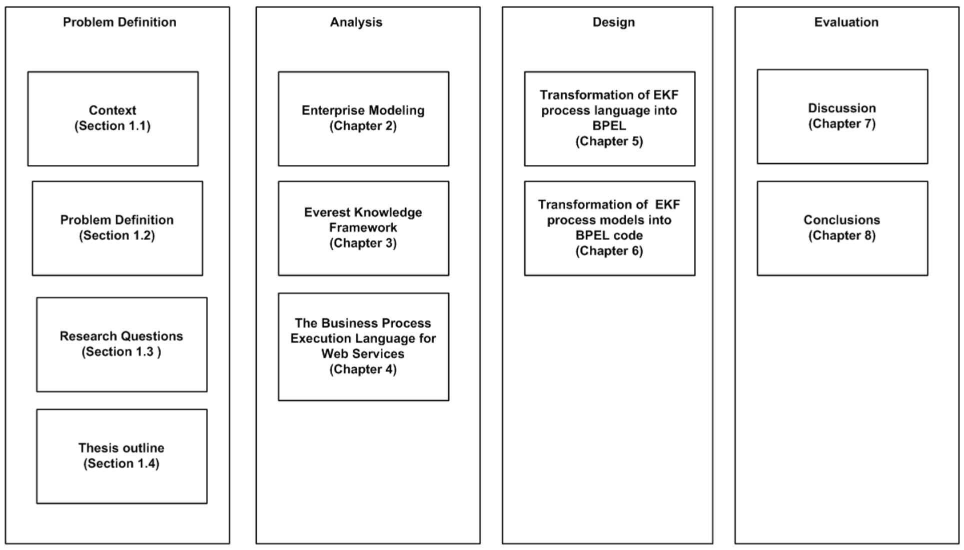

Problem Definition
Context (Section 1.1)
Problem Definition (Section 1.2)
Research Questions (Section 1.3 )
Thesis outline (Section 1.4)
Analysis
Enterprise Modeling (Chapter 2)
Everest Knowledge Framework (Chapter 3)
The Business Process Execution Language for Web Services (Chapter 4)
Design
Transformation of EKF process language into BPEL (Chapter 5)
Transformation of EKF process models into BPEL code (Chapter 6)
Evaluation
Discussion (Chapter 7)
Conclusions (Chapter 8)

FIGURE 1.2: Outline of this thesis.

In our analysis we start by describing the notion of enterprise modeling (see Chapter 2), because the first research questions involves identifying the principles of enterprise modeling as a strategy for development of business solution. We are mainly interested in enterprise modeling approaches applicable in the software development life cycle. We put enterprise modeling in the perspective of the EKF (see Chapter 3) to get more insight in the way enterprise modeling is performed in the EKF. Followed by a more detailed study of the way Everest models and deploys its processes in order to get more insight into the syntax and semantics of the EKF process language. We perform a similar study for BPEL to provide a more clear understanding of the key features, syntax and semantics of BPEL(see Chapter 4).

In the research design we focus on the approach followed to increase the interoperability of the EKF process design and implementation and aim at answering the fourth question: *'How to make the EKF more interoperable by the adoption of BPEL?'*. To provide a proper answer to this question, we respond to two partial questions, namely: *'What are the issues and challenges to close the gap between the EKF process language and BPEL?'* and *'How EKF process models can be transformed into BPEL code?'*. Answering these questions requires an assessment of the transformation of the EKF process language into BPEL. In our approach we translate EKF process language into BPEL by performing an analysis of the language contexts and language concepts, based on respectively the domain and technical spaces transformation (see Chapter 5). The issues and challenges which come from the domain and technical spaces transformation are important results, as they must be considered when proposing an approach to translate the EKF process models into BPEL code (see Chapter 6). By applying the proposed transformation approach to a case study (see Appendix D), we draw conclusions with respect to the correctness and completeness of our proposed transformation solution.

In the evaluation we reflect our solutions, for which we aim answering the last question: *'What are the best practices which must be considered when adopting a BPEL engine in the EKF?'* (see Chapter 7). To answer this question we first: compare the EKF process language with alternative modeling languages, second: we start a discussion of the best practices which must be considered when integrating a BPEL engine in the EKF, third: we give an overview of the related work, and finally: we discuss the limitations of our proposed solution. The conclusions are summarized in chapter 8, followed by the bibliography.

#  Chapter 2

# Enterprise Modeling

In this chapter we aim at answering the question: *'What are the principles used for enterprise modeling?'*. Therefore we start with an introduction to enterprise modeling (see Section 2.1), followed by a more detailed description of enterprise architecture (see Section 2.2) and MDA (see Section 2.3). Both enterprise architecture and MDA are considered to be important approaches in the context of the application development life cycle of Everest (see Section 2.4).

## 2.1 What is Enterprise Modeling?

When new business applications are being developed, different kind of stakeholders are involved. Each stakeholder has its own perception of the business applications and evaluates the problem from different angles, also referred to as 'viewpoints'. The viewpoints can be represented by a vocabulary that specifies a model, which can be used for communication and synchronization purposes among the different stakeholder. Different models are used to fulfill the need of the different stakeholders (e.g. scope, business model, system model, technology model and detail representation). Each viewpoint has several aspects, particular facets that need to be considered to completely describe each viewpoint. The aspects are based around six basic questions as defined by [Zachman, 1987]: what (data), how (function), where (network), who (people), when (time) and why (motivation).

Enterprise modeling is *'the activity that is used to create abstractions of models that captures different aspects of the business with a purpose to understand and share the knowledge of how the enterprise is structured and how it operates'* [Bajec and Krisper, 2005]. A model is *'the abstraction of its subject that includes information pertinent to* *its viewpoint and omits information to other viewpoints'.* A model can be decomposed into individual models that together describe all aspects of all viewpoints. Different viewpoints can have common elements, but relations to other viewpoint can exist which make the models interdependent. Different enterprise models are proposed to express the different aspects of the business: business motivation model, business process model, domain concepts model and organizational model.

The motivation model reflects the strategy, goals, policies, laws and regulations from the business perspective and need to be made explicit in terms of the business rule model. This is referred to as the *'why'* of the business operation. The *organizational model* reflects the organizational structure and the resources that are important for the organization, which reflects the *'who'* of the business operation. The *business domain concepts model* captures the concepts (e.g. products, services and documents, etc.) that a business maintains, and represents the *'what'* of the business operation. The *process model* reflects procedures an organization maintains, describing in which order the work need to be performed to reach a certain business goal. The process model ensures certain efficiency of the business and is focused on the *'how'* of the business. The enterprise models do not operate independent of each other, such that alignment between the models is required through the business model, before they can be put to actual use [Bajec et al., 2000, Atkinson and Kuhne, 2003].

Both the *enterprise architecture* (see Section 2.2) and *MDA* (see Section 2.3) are interesting in the context of Everest, as they are the basis of the way Everest models and deploys business applications as part of the Everest development life cycle.

## 2.2 The Enterprise Architecture

The enterprise architecture is the practice of optimizing and aligning the organizations, decisions, processes and services a business maintains. It consists of describing the current and future structure and semantics of the: processes, information systems, personnel and units, and how they are aligned with the organizations strategic direction. In this thesis we are mainly interested in the technology aspect of the enterprise architecture, which is focused on alignment of the functional components of an actual business application. The technical aspects consider logical components to be a decomposition into systems, which are responsible for controlling these aspects.

Figure 2.1 shows a conceptual overview of a generic enterprise architecture composed of the following five layers[1]:

[1]Notice that the layered stack is extended with event and rule layer as an independent system decomposition.

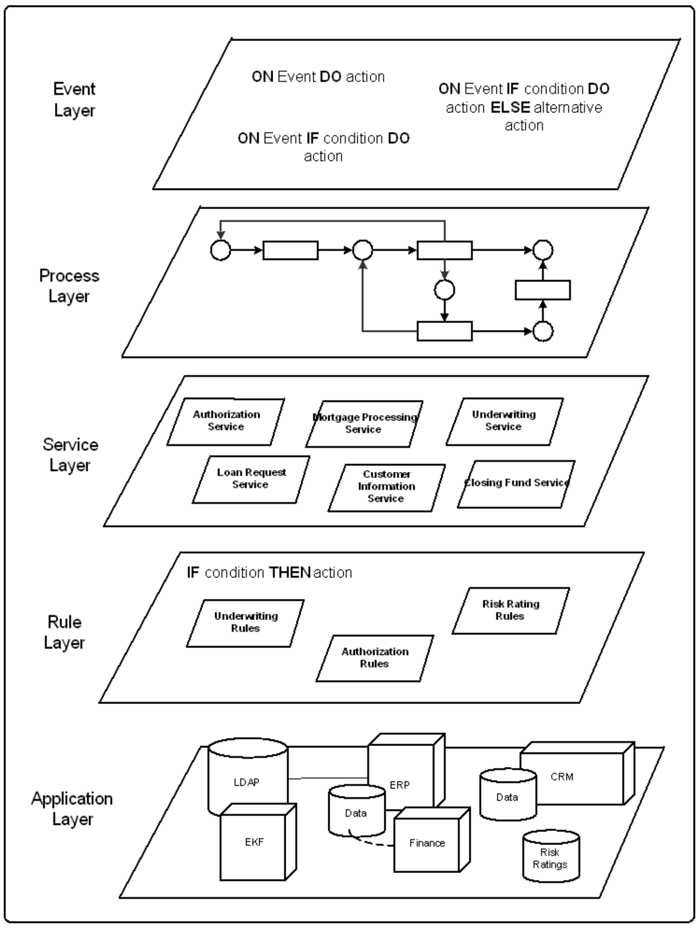


FIGURE 2.1: Layers in the enterprise architecture [Wilkes and Veryard, 2004].

- The *event layer* exposes a predefined set of event patterns as services, used for specifying communication protocols or the processing of (internal and external) business events.
- The *process layer* combines the services in long running processes, where each process itself is provided as a service.
- The *service layer* conceals the system functions and components and provides them as services.
- The *rule layer* can be used to describe the structural and semantical constraints of the actual service.
- The *application layer* is responsible for the integration of services across application systems (e.g. Enterprise Resource Planning, Customer Relationship Management, etc.).

Instead of implementing each enterprise applications as a single purpose application, one can use architectural styles as a blueprint for the design and implementation. Architectural styles promotes reuse as they promote the modularity of components across the enterprise architecture. Take for example a jigsaw puzzle is modular, but it can only be composed in one way (close-ended). A tangram is a tiling puzzle that is also modular, but can be composed to make an infinite variety of shapes (open-ended). Following the tangram approach each layer in the enterprise architecture, promotes a style reflecting one functional component in the enterprise architecture. The following architectural styles can be identified: Service Oriented Architecture (SOA), Event Driven Architecture (EDA), Process Driven Architecture (PDA) and Rule Driven Architecture (RDA) [Kuster and Konig-Ries, 2007, Michelson, 2006, Martin, 2006]. In the following sections we give a brief introduction to each of the architectural styles.

### 2.2.1 Service Oriented Architecture

The SOA is *'a computer systems architectural style for creating and deploying packaged services and defining an infrastructure to allow the interaction of these services'*. The services are reusable pieces of functionality that communicate with each other in some meaningful way. The interface of the services are described by an universal language, such that they can be invoked from applications or services independent of the platform on which they operate and language in which they are implemented.

The SOA prescribes a mechanism to provide and consume services in a network. This allows that the provided services can be reused by multiple consumers. Reuse of the services reduces complexity when multiple parties need to interact with a single service. A system which has adopted SOA consists of subsystems on the application layer that interact in a loosely coupled manner. Loose coupling describes: *'an approach where integration interfaces are developed with minimal assumptions between the sending/receiving parties, thus reducing the risk that a change in one application will force a change in other application*[2]*'*. The subsystems are autonomous, such that they can be consumed by different partners without knowing the processing details of the actual service. Services are subject of reuse as they are decomposable (service can contain services). Another important characteristic of services is that each service can be treated as a black box, such that the consumers of services do not require a detailed definition of how the service was implemented, but only the (semantic) specification of the service. Finally, services increase scalability as each service should be dynamically discovered and bound to the end-point, independent of the location or platform on which they operate. Different communication protocols (e.g. CORBA, RMI, SOAP, COM, etc.) have been proposed,

[2]see: http://en.wikipedia.org/wiki/Loose_coupling

promoting implementations of SOA across different type platforms (e.g. Sun Microsystems (J2EE) and Microsoft's (.NET)).

### 2.2.2 Event Driven Architecture

The EDA can be defined as: *'an architectural style for designing and implement applications and systems in which events transmit between loosely coupled software components and services'* [Michelson, 2006]. An event driven system is typically comprised of event consumers and producers. Event consumers subscribe to an event provided by an event manager, the event producers publish the occurrence of the events to this manager. When an event is received by the event manager it is forwarded to all subscribed consumers. In case the consumer is not available, the manager can store the event and try forward it at a later stage (store and forward).

EDA is typically useful for business-to-business and peer-to-peer interactions which are highly dynamic across business applications. Therefore a more sophisticated a-synchronous processing technique is quickly becoming apparent to support this task. Building a communication model to exploit this power and allowing a certain flexibility is a high priority for competitive software development. An event-driven communication model is superior when responding to run-time occurrences, compared conventional synchronous request/reply mechanism. As the publisher has no knowledge of the events subsequent processing or the interested parties, it is clear that the events are by nature loosely coupled and highly distributed.

EDA is push-oriented and fills the gap of PDA which is generally pull-oriented. Take for example a mortgage company detecting an increasing number of fraud situations for which different controllers need to be notified.

### 2.2.3 Process Driven Architecture

The PDA can be defined as: *'an architectural style for designing and implementing applications and systems in which processes are exposed and interact between loosely coupled software components and services.* The origin of PDA[3] comes from the workflow management paradigm, which is focused on *'the division of work, such that it can be distributed among the different actors in the process'.* In the typical workflow paradigm, a business process is recursively decomposed into sub-processes and tasks, where the task is an atomic entity. Here also lies the main difference between workflow and PDA,

[3]There is no agreement on the meaning of PDA, but there are topics with respect to business processes that are commonly gathered under this terms, notably the design, analysis, modeling, implementation and control of the business processes.

as PDA exposes the atomicity of processes, sub-processes and tasks as services. This principle is also referred to as the *orchestration* of services.

Workflow is related to three dimensions as identified by [Aalst van der and van Hee, 2004]: control flow, case data and resources. The *control flow* defines the order in which the task must be performed; the *case data* defines the goal in form of information that need to available after the completion of each task; and the *resources* refer to the organizational structure such as: actors and their capabilities and responsibility to perform tasks. The dynamic behavior of the process is conceptualized in terms of a case following through the tasks and sub-processes according to a predefined process model. A case can follow alternative choices or parallel-split routes through the process model, where the split implies that the case can reside at several positions in the flow simulations. A more elaborate description of the variations in control flow routing are is described in the Workflow patterns [Aalst van der et al., 2003, Aalst van der and Hofstede ter, 2002, Puhlmann and Weske, 2005].

Workflow has introduced various advantages with respect to modeling the procedures in the operational business. The separation of control of work, can be distributed among human and system actors and even third participating parties in the business environment. In this way workflow decreases the operational cost by automating the manual tasks (e.g validate, retrieve and interchange of data on a timely basis.). Workflow enables optimization of the business performance (e.g. decrease throughput time and waiting times in the process), by performing measurements, analysis and redesign [Reijers and Limam Mansar, 2005]. By modeling the process instead of programming it is possible to validate its correctness[4] or simulate the process before the they are put in actual use [Aalst van der and Hofstede ter, 2002, Aalst van der, 1998, 2000, Jensen, 1997, Dijkman et al., 2007]. Workflow also contributes to the compliance in the organization, such that a process can be used as an agreement between multiple parties [Zaha et al., 2006]. Workflow solutions are better adaptable to changes from the business, because the process is defined in a graphical language, understandable by business engineer. Changing a graphical language instead of the programming logic increases the agility to adapt to changes and decreases the time-to-market.

Drawbacks associated with the workflow paradigm are lack of flexibility, the atomicity of tasks, context tunneling and the mix-up of distribution and authorization [Aalst van der et al., 2005b]. The *lack of flexibility* refers to the push-oriented routing focusing on what should be done instead of what can be done, resulting in rigid inflexible workflows. The *atomicity of tasks* refers to the requirement that work should be straight-jacketed into atomic tasks, while user performs them at a much more fine-grained level. *Context*

[4]Only possible when the semantics can formally described.

*tunneling* refers to the lack of context during the processing of a task by an individual, especially the processing history of a case. The *mix-up between distribution and authorization* refers to the problem that workers can see all the work they are authorized to do, but are not authorized to do anything outside of their worklist.

### 2.2.4 Rule Driven Architecture

The RDA can be defined as: *'an architectural style for designing and implementing applications and systems in which the semantics of a set of rules are exposed as loosely coupled software components and services'*. The rules are declarative statements expressed in a basic syntactical structure. The RDA encompasses improving the correctness, consistency, changeability and agility of the decision taking an organization maintains. More specific: RDA exposes the decision taking as services for which the semantics are defined in terms of declarative rule statements.

The concept of business rules originates from the knowledge engineering[5] and knowledge representation[6], which both have arisen from the cognitive science and artificial intelligence. In these paradigms the knowledge is used to achieve intelligent behavior as to facilitate inference (i.e. taking decisions or derive knowledge from explicit knowledge). Rules are stored in a rulebase and are enacted by rule engines, which evaluate the conditions of the rules and determine (at any point in time) which rules are eligible to fire. Newell [1982] is one of the first that introduced the idea of knowledge modeling on the conceptual level, his work states that: *'knowledge is to be modeled at a conceptual level, in a way independent of specific computational construct and software implementation'*. Other considerable research comes from the Database Research Community, which has resulted in the development of active databases supporting data integrity, through procedures and triggers [Dayal et al., 1998, Tanaka, 1992, Widom and Ceri, 1996]. Further research on rules has resulted in the development of deductive databases, which make deductions based on rules and facts stored in a rulebase [Loucopoulos, 2000, Loucopoulos et al., 1991]. These approaches have evolved into the rich knowledge based systems we use today, where rules are separated from the data and processed by independent engines [Ross, 2000, Schreiber et al., 2000]. The Business Rule Approach is the result of the increasing interest of identifying and articulating the operational business rules in a development life cycle [Ross, 2003]. The research community further investigates the elicitation, analysis, classification, formalization and implementation of business rules [Herbst, 1994, Rosca et al., 1995, Bajec et al., 2000]. The more common use of rules has resulted in the development of vocabularies and standards for the interchange of

[5]see: http://en.wikipedia.org/wiki/Knowledge_engineering
[6]see: http://en.wikipedia.org/wiki/Knowledge_representation

rules by means of services. Nevertheless, at the time of writing the development of these standards is work in progress [OMG, 2006, W3C, 2005].

The Business Rule Manifest [Ross, 2003] specifies a number of guideline, which need to be followed in order to take advantage of business rules. Traditional software engineering is a slow and time consuming process, where the software (which will end up in the programming logic) implicitly describe the executable rules. The business rules aim at separation of the executable rules from the programming logic, by implementing the rules as content [Herbst, 1994, EnixConsluting, 2005]. Like the process the business rules increase the compliance in the organization as the rules explicitly define the policies & regulations an organization maintains. The business rules are an important asset of the business (reflecting the requirements of the business), but with traditional system development these rules are buried deep into the programming logic.

The main difference between the rules and the process is that rules are modular and have a declarative syntax compared to the imperative syntax of procedural processes. The rules are most useful at a more fine-grained level of the process, where it becomes increasingly difficult to express the business knowledge in terms of imperative languages. The rules are modeled as independent rule statements, such that each rule can be seen as the smallest unit of change [Goedertier and Vanthienen, 2006]. The changes of business rules can be simulated without changing code, but by changing the rules instead [Bajec et al., 2000, Bajec and Krisper, 2005]. Analysis can be performed at forehand to evaluate if the newly implemented business rules meet the actual business objectives. The declarative nature of rules allow a better understanding of the individual rules, such that they can be modeled by the business engineers and allows validation of independent rules more easily. By making business rules explicit and express them in syntax understandable for business engineers, brings the rules one step closer to the business to regain control of its rules. The software engineers do not have to translate the rules into programming logic, but develop a rule engines that are responsible to enact the rules.

## 2.3 Model Driven Architecture

Another approach of enterprise modeling can be found in the MDA, which can be defined as: *'a model based representation of a part of the function, structure and semantics of an application or system'* [Mellor, 2004]. MDA is not focused specifically on the functional architecture, but more on the modeling perspective. MDA classifies three model types: Platform Specific Model (PSM), Platform Independent Model (PIM) and Computational Independent Model (CIM). MDA proposes high-level modeling languages, which introduces a more abstracted modeling syntax reducing technical complexities.

These models are referred to as CIM, which allows modeling independent of the actual implementation, where the solutions will be limited by technical considerations (e.g. BPMN, AD and YAWL, etc.). In MDA the models for which the platform independent computation are considered are referred to as PIM (e.g. BPEL, XPDL, RuleML, etc.) and the models which include platform specific specifications are referred to as PSM (e.g. C# and Java, etc.).

At all levels of MDA models are governed with a syntax and semantics[7]. This allows that models, for which the syntax and semantics are comparable, can be synchronized through transformations from CIM into PIM and from PIM into PSM. To accomplish this goal OMG [2003] has defined four modeling layers in MDA (see Figure 2.2). At the M0 level, MDA specifies the executable code of a program. At M1 level, MDA considers models from which the executable code can be constructed. At the M2 level, MDA specifies the meta-models that define the syntax of the language used for modeling. The M3 level, MDA specifies the meta-meta-model that defines a generic language, which can be used to construct meta-modeling languages at the M2 level of MDA[8].

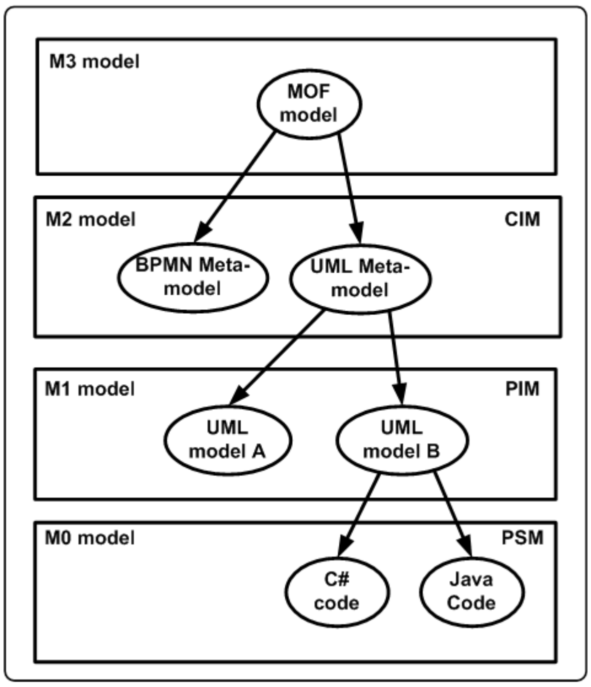


FIGURE 2.2: Modeling layers in MDA from [Kleppe, 2003].

The goal of MDA is to transform models from one modeling layer into another[9]. An alternative approach to pure MDA, is to define engines, which interpret the models at M1 and produce the desired behavior based on a set of predefined semantics. In this

[7] A model is formal if its syntax and semantics can be described formally.
[8]In figure 2.2 Meta Object Facility (MOF) is provided by OMG to fulfill this task.
[9]In this thesis we mainly focuses on the forward engineering principle, the backward engineering is not considered.

case it is not required to generate programming logic, but generic engines implement the semantics of language concepts at the M0 level of MDA. This allows that the models can be simulated at an early stage in the development life cycle, unlike the traditional approach where programming logic drives the solution. MDA also ensures a certain predictability of the end result and encourages efficiency of system models in the software development life cycle. MDA also supports reuse of best practices when creating families of systems, allowing the use of automated tools for the design and validation of models and facilitate the transformation of models at different levels of abstraction.

## 2.4 Enterprise architecture and MDA in the life cycle

The life cycle plays and important role in enterprise modeling to enable different stakeholders to manage their own viewpoints of a business solution. In this section we discuss the role of the enterprise architecture and MDA in the application development life cycle. A general development life cycle is composed of generally four repeating stages (see Figure 2.3): design, implementation, enactment and analysis.

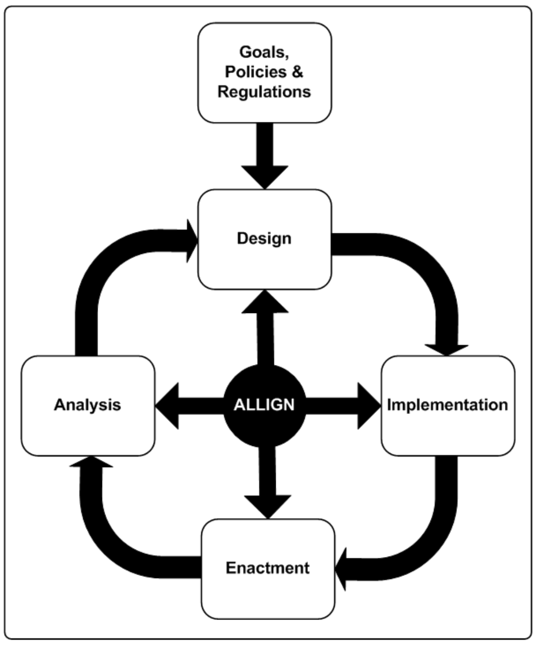


FIGURE 2.3: Stages in the development life cycle (from Muehlen zur [2004]).

### 2.4.1 Design Stage

At the design stage the business goals, policies and regulations (also known as business requirements) are made explicit in terms of business requirements. These business requirements must be translated into designs, which is the first step toward a more formalized representation of the business solution. The business requirements are dynamic and should be flexible toward changes in the business. Changes in the business environment are driven by internal decisions or external forces, such as government laws and regulation. The regulation imposes external legislations, such as business protocols, legislation, long-term contracts, quality norms etc. Regulations often bring change regarding the process interaction with the business partners. The policies are internally defined and come from intended strategies of management or procedures for the personnel. Take for example: an order acceptance policy, discount policy or risk rating policy. The changes to the policies are motivated by strategic, tactic and operational decisions. The strategic and tactic policies are not formal and need to be formalized into operational policies, which can be expressed in the form of a design governing by a syntax of the form: principles, procedures, facts, figures, formulas, etc.

The design stage is driven by business-oriented people, which are mostly not familiar with formal modeling languages. At this stage modeling is performed at the conceptual level, classified as CIM in MDA. No technical aspects of the enterprise architecture are considered yet at this stage. Nevertheless, the organizational models describing the geographic decomposition of the organization, already constraints aspects of the enterprise architecture.

### 2.4.2 Implementation Stage

Business-oriented people have little understanding of programming languages, such that involvement of business engineers and software engineers is required at this stage to translate the design into an actual implementation. Unfortunately, this approach leaves the business user with very little empowerment to manage and control the actual implementation, because after the implementation the programming logic or executable models hide the business requirements.

The implementation requires that the design is formalized at a more fine-grained level. We can classify these models as PIM or PSM in MDA. These models are either implemented as programming logic or executable models. Using formal models instead of programming enables validation at an early stage of development. The technical architecture specifies that actual targeted platform on which the executable models or programming logic operates. The technical consideration at this stage directly affect the selection of the platform or the way the business engines are implemented. Technical limitations arise as a result of these technical considerations which need to be resolved for the design (e.g. resolve language concepts in design not supported by the implementation). This implies that the decomposition of components of the enterprise architecture need to be accomplished before or during the actual implementation.

### 2.4.3 Enactment Stage

During the enactment stage the models (result of the implementation stage) are deployed on the run-time environment. The run-time environment is the platform on which the programming logic operates or the implementation of the business engine (responsible to enact the executable models). The programming logic and executable models are referred to as PSM in MDA, as they are translated from an implementation into an operational business application. The deployed applications and the run-time environments are part of the enterprise architecture.

### 2.4.4 Analysis Stage

During the analysis stage the goals, policies and regulations are evaluated and monitored to determine if they have been met (e.g. performance, efficiency, etc.) by the operational application. Either the violation or change of goals, policies and regulations could result in change of the design, implementation and enactment. This included the validation of the correctness and completeness of the actual implementation. Validation at this stage is required, because programmers often disperse programming logic into pre-existing implementations. Analysis could require that the design and implementation are re-examined and modified according to the new objectives and goals of the business.

At the analysis stage models define how the actual evaluation must be performed and who is responsible to take actions in case certain goals have been violated. The enterprise architecture specifies the locations, where the information, required for the analysis can be retrieved.

In this chapter we have presented that enterprise modeling is an approach for creating models for different viewpoints of the business applications. Both enterprise architecture and MDA have emerged as principles of enterprise modeling. The enterprise architecture focuses on the functional decomposition of a system into components. MDA specifies how modeling is performed, and allows transformation of these models at different levels of abstraction. The enterprise architecture and MDA are important in the context of the way Everest models and deploys business applications in the EKF. In the following chapter we therefore continue to discuss how enterprise modeling is specifically applied in the EKF.

# Chapter 3

# Everest Knowledge Framework

In this chapter we aim at answering the question: *'How does Everest apply enterprise modeling in the EKF?'*. Answering this question is important as Everest has implemented a framework for developing business applications, but limited documentation exist that provides a more detailed overview how enterprise modeling is applied in the EKF. In this chapter we therefore give a more clear understanding into the way enterprise modeling is supported by the EKF (see Section 3.1), which is considered to be an important result for Everest. The EKF is a modeling and enactment environment based on the enterprise architecture and MDA approaches, as presented in the previous chapter. We discuss how the enterprise architecture and the MDA are covered by the EKF to accomplish the development of business applications from design into the actual implementation (see Section 3.1.1 and 3.1.2). This includes a more detailed study of the EKF process modeling language as plays a central role in the preceding of this thesis (see Section 3.1.2.3). Finally we discuss how the EKF process models are actually deployed on the enterprise architecture of the EKF (see Section 3.2).

## 3.1 EKF and Enterprise Modeling

The EKF is a framework used for the development of knowledge intensive (front- and mid-office) applications. Everest has adopted enterprise modeling as a strategy for developing business applications. MDA plays an important role in the development of a set of models which together describe business application. For modeling, Everest considers conceptual models at design-time, which are translated into executable models. The executable models are deployed on business engines, which are considered to be part of the functional components of the enterprise architecture. In the following sections we give more insight into of the enterprise architecture of the EKF, followed by a more detailed study of how MDA is specifically performed in the Everest modeling approach.

### 3.1.1 Enterprise architecture of the EKF

The enterprise architecture of the EKF is composed of a number of logical components including business engines and additional systems performing specific aspects in the EKF. At design-time the models are implemented using the Everest Knowledge Studio (EKS), from which they are deployed on the targeted business engines. Each logical component specifies the semantics of the executable models at run-time. The following logical components are supported by the EKF (see Figure 3.1): domain storage, transformation engine, portal engine, event manager, rule engine, process engine, worklist manger, and Enterprise Service Bus (ESB).

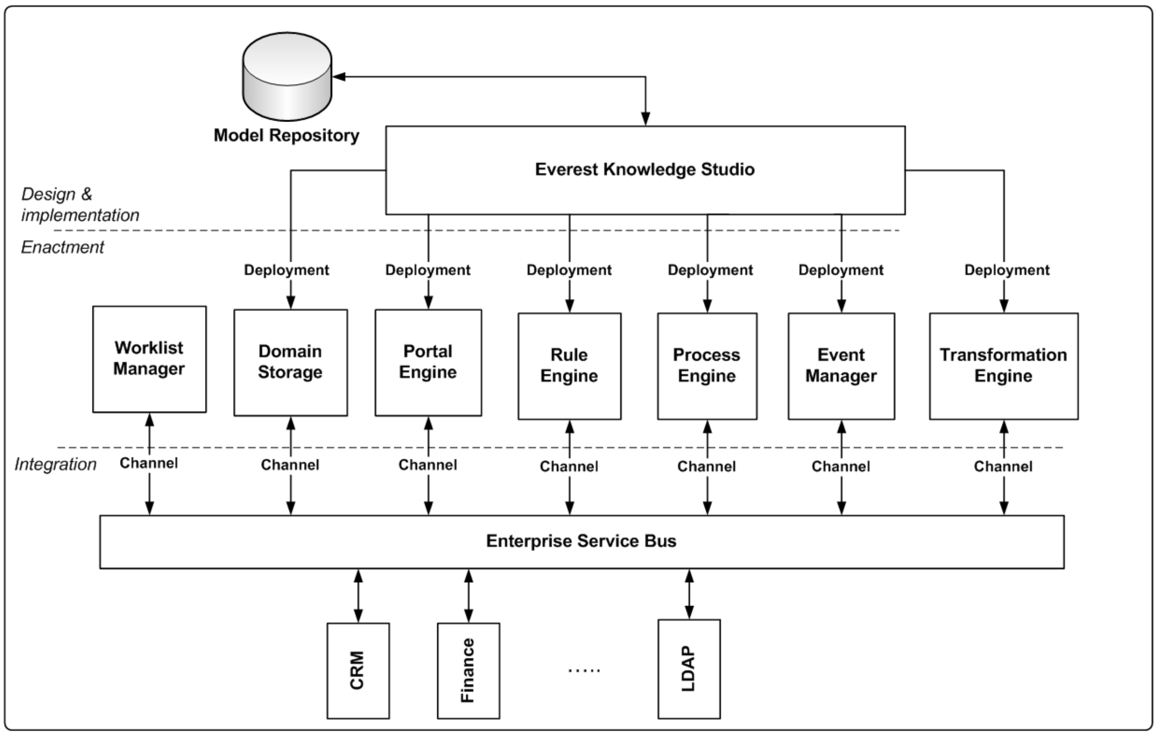


FIGURE 3.1: Enterprise Architecture in the EKF.

**Everest Knowledge Studio**
The EKS provides an environment, where the models can be implemented and deployed on the engines. The models are stored in an extensible model repository from which the different versions of the models are maintained.

**Domain Storage**
The domain storage maintains persistence, consistency and integrity of the domain data. The domain models are deployed on the domain storage, from which instances of the

models are instantiated and maintained. The domain storage is an abstraction of the actual data storage (database) and exposes the data in terms of business objects instead of relational database structures. The domain storage allows queries on the domain data, such that application data is maintained and retrieved through the domain storage component.

**Transformation Engine**
The transformation engine is responsible to transform one data format into another data format. Transformation of data is specifically useful in case system-to-system integration requires various data formats to be published and consumed.

**Portal Engine**
The portal engine uses a presentation model to drive the semantics of the user interaction through a dialog interaction with the user. The behavior of the user interactions are dynamically generated, from the presentation model. The presentation model defines the semantics, affecting the user experience independent of the way its presented. The presentation model allows the modeling of the dialogs and services in terms of an application flow. This allows the realization of dialogs and navigation, such that business-oriented people can develop these dialogs based on attributes from the domain model. The dynamic behavior of the dialogs is controlled by the rule engine and the presentation is generated through the transformation engine.

**Event Manager**
The event manager is responsible for managing event-driven aspects in the EKF. The event manager is primarily an event processing concept that deals with processing multiple events. The event manager provides a platform for interaction between the other components (e.g. portal engine, rule engine, process engine, etc.), such that each component can publish and subscribe to events. All components subscribed for a certain event are notified after the occurrence of that event.

**Rule Engine**
The rule engine interprets the rules and derives knowledge from the domain data. In traditional computer programming, a program is composed of an algorithm and a data structure. A rule based system uses a rule engine to derive knowledge by inference rules included in the rule repository. The EKF supports backward chaining as a strategy for the rule inferencing. Backward chaining starts with a list of goals or hypothesis and works backwards to see if there are data attributes available that will support any of these goals. A rule engine using backwards chaining, searches typically for rules for which the *then clause* matches the desired goal and adds it to a list of goals if the *if clause* is known to be true. In order to realize the goal the data that confirms these goals must be provided.

**Process Engine**

The process engine is responsible for the execution of the process specific aspects and maintains the information of the control flow in the process. The process engine relies on both the event manager and rule engine to determine when a certain action must be performed. The process engine itself specifies, the order and through which channel the actions must be performed. More specific the process engine is the central controller and coordinator across the logical components involved in the business process.

**Worklist Manager**

The worklist manager is responsible for the distribution of work between the (system and human) actors in the process. The worklist manger considers some kind of strategy to determine *who* (actor) is responsible to perform the task. Roles are associated with worklists (also called work queues or in-boxes), in which a workitem appears as soon as a case is ready for processing. Users can pick-up a workitems from the worklist associated with their roles, which result in the start of an application to complete the task corresponding to the workitem. Tasks can be automated if they are governed with a system roles, such that they are performed through a self contained application. Another responsibility of the worklist manager is to prioritize workitems in the worklist and specify the strategy in which the tasks are provided to the actor (push or pull mechanism)[1].

**Enterprise Service Bus**

The Enterprise Service Bus (ESB) enables the integration and communication with external systems and components through different channels. The service logic conceals the external applications, by providing services from which they can be invoked. The service logic interacts with application layer by means of channels[2]. The EKF does not support a model for specifying the service interaction, but programming is still required to accomplish this property. The interaction is therefore implemented as modules in the EKF supporting the interaction with different system-to-system protocols, which promotes the interaction across different applications operating on different platforms (e.g. JMS, EJB, IBM-MQ, HTTP, SOAP etc.).

### 3.1.2 Model Driven Architecture in the EKF

In the development approach of Everest, graphical models are used at the design stage, mainly for the communication with the different stakeholders. The graphical models are implemented in terms of the EKF executable models. These executable models are interpreted by the business engines, which specify the semantics at run-time.

[1]In the EKF this strategy can be configured by means of rules.

[2]The EKF channel is the service of the EKF enterprise architecture

The EKS is used for construction of respectively five types of executable models (see Figure 3.2): domain model, rule model, task model, process model and interaction model. These models can be used for defining models to describe the system behavior, independent of how they are enacted by the underlying logical components. Each model is interpreted by the logical component from which the desired behavior is generated at run-time.

The *domain model* underlies all models, specifying the structural aspects of the applications. The *rule model* describes the dynamic aspects in terms of rules. The process, task and interaction model are implemented in terms of a combination of the domain and rule model. The *process model* specifies how control flow of tasks in the process and how they are distributed among the different actors in the process. The *task model* specifies how the goal in the process is accomplished in terms of a human-to-system or system-to-system interaction at a more fine-grained level. The *interaction model* specifies how the human-to-system interaction takes place through a navigation of dialogs. The *task model* and *interaction model* are outside the scope of this thesis. Therefore we deal with tasks as synchronous channels provided by the EKF connectivity layer, which require information on its input and provides information on its output when completed.

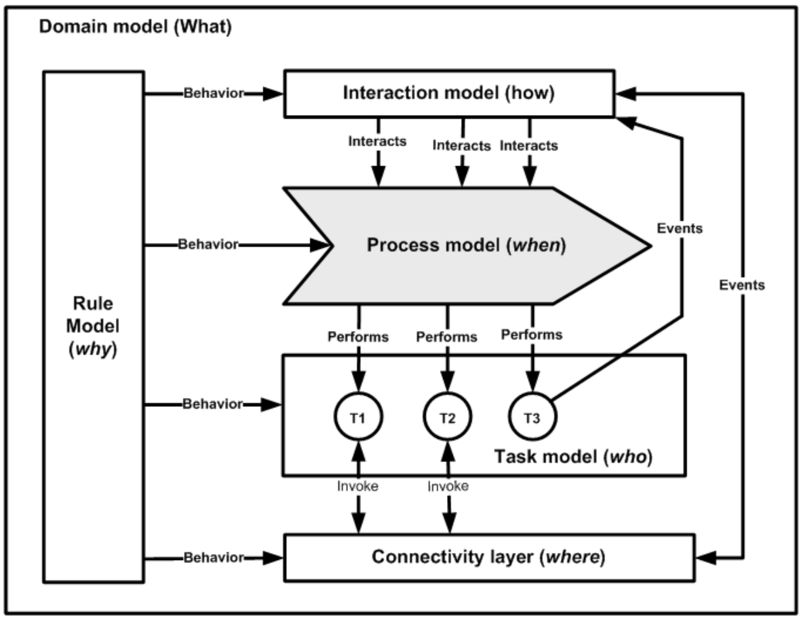


Figure 3.2: Modeling dimensions in the EKF.

The basic principle of the EKF is to separate the storage of domain and rules (content) from the engines (programming logic). Everest uses a graphical languages for modeling at design-time. These graphical models are used as a guideline to implement certain aspects in the EKF in terms of the domain and rule models.

The domain and rule languages are therefore considered to be modular as they are not specifically designed to model a specific aspect of the system, but allow modeling an extensive number of shapes (concepts). The domain model can be used for modeling the business domain or to construct meta-models for modeling languages. The rule model is used for data validation, derivation of knowledge from existing knowledge and to accomplish the dynamic aspects in the EKF by making certain semantics explicit in terms of rules. The process model inherits properties of the domain and rule model, as the process language actually implemented in terms of domain and rule models. More specific, the structural aspects of the process are implemented in terms of a domain model and the behavioral aspects (e.g. state transition, decision taking) are implemented in terms of a rule model.

Both the domain model and rule model are part of the process implementation in the EKF and are required to completely understand the syntax of the EKF process. The domain model and rule model are interpreted by respectively the domain storage and rule engine, which specify some of the semantics in the process. In extension of the domain and rule engine, a process engine is introduced to accomplish some of the process specific semantics of the process. In the following sections we continue to provide a more detailed description of the syntax of the domain model, rule model, process model and connectivity layer.

#### 3.1.2.1 Domain Modeling

The domain language captures the structural aspects in terms of entities, attributes and relations, which is similar to Unified Modeling Language (UML) [OMG, 2005]. The domain language can used at the M2 level of MDA to construct meta-models of a modeling language (e.g. process model, rule model, etc.) and on the M1 level of MDA to model business concepts (e.g. organizational model and product model, etc.). An example of the domain model is included in the case study (see Appendix D).

In figure 3.3 we present the UML meta-model expressing the syntax of the EKF domain model. The EKF domain language supports two types of relations: association and generalization. The *association* relation is comprised with cardinality relations between two entities and *generalization* allow inheritance and specialization of entities. *Entities* maintain a number of attributes and optional *operations*. *Attributes* in the domain model support the following basic data types: *integer*, *double*, *float*, *boolean*, *datetime*, *string* and *enumerations*. Each domain model maintains instances of domain data, for which attributes refer to the values assigned to attributes. Notice that multiple instances of the domain data for a single domain model can be maintained. The *integrity rules*

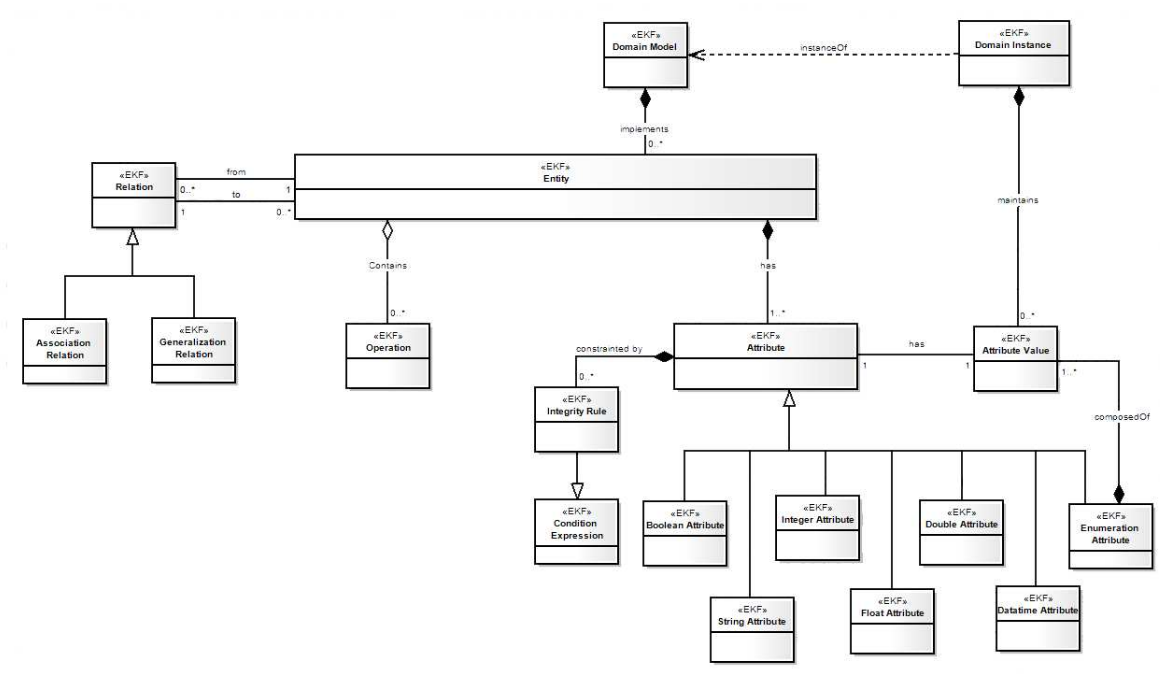


FIGURE 3.3: Meta-model of the EKF domain language.

underlie the domain model and guards the integrity, persistence and consistency of the domain data. The integrity rules do not prescribe a dynamic aspect of the system, but constraint the domain model throughout its existence.

#### 3.1.2.2 Rule Modeling

The rule language allows defining the rules instead of programming. The rules are strongly related to the domain model, as the conditions are composed of domain attributes and the actions are governed by the assignment of values to domain attributes or invocation of operations. The goal of the rules is to derive knowledge (in terms of domain attributes from the domain model) from existing knowledge or perform predefined actions through operations.

The rules are implemented as executable rules, which are part of the Event Condition Action Alternative Action (ECAA) paradigm. This paradigm originates from the active database field, where ECA rules are implemented as database triggers. The event specifies some temporal behavior in terms of events; the condition is an expression composed of terms and fact from the domain model; the action is a procedure or operation that needs to be performed; and the alternative action must be performed for the set of alternative conditions. The EKF rule language allows: Event Condition Action (ECA), Event Action (EA) and Condition Action (CA) rules to be expressed in terms of either decision tables, decision trees or independent rule statements. Wagner [2002] has identified the following classification for executable rules, which is based on the syntactical structure of the rules:

- The *'integrity rules'* are described as constraints on facts or associations of two or more terms. Notice that the integrity rules are supported by the EKF domain model and are not part of the rule model. Example: *the customer must be at least 18 years old.*

- The *'production rules'* are rules which specify that in case a certain condition holds a corresponding action needs to be performed. Production rules are mainly supported by forward chaining rule engines. Production rules are not supported, as the EKF rule engine only supports backward chaining inferencing. Example: ***if*** *the customer credit is greater than 10.000* ***do*** *calculate the discount*.

- The *'derivation rules'* are the statements of knowledge that are derived by explicit knowledge through: inference of rules, logical expression or mathematical calculation. Derivation rules are processed by either a forward or backward chaining rule engine. The EKF only supports backward chaining mechanism to process derivation rules. Example: ***if*** *the monthly debt divided by yearly income is greater than 30 percent* ***then*** *the customer qualification is accepted* ***else*** *the customer qualification is rejected*.

- The *'reaction rules'* are described as actions performed in response of some event. Reaction rules are supported in the EKF through the event manger. Example: ***on*** *the occurrence of a risk exception and* ***if*** *the loan was accepted* ***do*** *send notification message*.

- The *'transformation rules'* are the rules used for the transformation of one data format into another data format, which is directly supported through EKF transformation engine.

Three rule representations have been adopted in the EKF rule language: rule statements, decision trees and decision tables. A *rule statement* is an expression of a single rule independent of the other correlated rules. A *rule expression* is expressed in terms of: derivation rule (**if** condition **then** action) or reaction rule (**on** event **if** condition **then** action). A *decision table* is a tabular representation to describe and analyze decision situations, where a number of conditions determine the assignment of a set of attributes. Not just any representation, however, but one in which all distinct situations are shown as columns in a table, such that every possible case is included in one column. A *decision tree* is a graphical representation of a ruleset represented as tree of rule conditions, for which each leaf of the tree specifies a specific action. The decision table and decision tree guard the properties of completeness and exclusivity as described in [Vanthienen, 1991, Kriz et al., 1998]. Additional symbols are adopted in the EKF rule language to accomplish the following semantics:

- *Hypen* $(*)$, acts as a don't care, which can be read as true.
- *Unknown (?)*, evaluates to true if a certain data values is unknown at a certain point in time.
- *Else* $([])$, indicates all other possible condition alternatives.
- *Empty* $(-)$, indicates that no additional events or conditions are considered or actions are performed.

An example of the rule model (rule statement, decision table and decision trees) is included in the case study (see Appendix D).

Figure 3.4 gives an overview of the syntax of the EKF rule repository expressed in terms of an UML meta-model. The root element of the rule language is the *ruleset*, which is a grouping of rules. A ruleset is composed of *decision tables*, *decision trees* and *rule statements*. A ruleset is a sub-set of rules which are correlated by means of their common conditions and actions. A ruleset therefore defines the total set of rules that specifies the action taken considering all possible conditions and optional events. A ruleset contains all correlated rules, which need to be executed at a common point in time. The rules refer to other rules at the time of their execution, due to fact that rules can use actions of preceding rules as a condition. Optional *event* triggers can be defined, such that a certain ruleset is only evaluated after the occurrence of this event. For each rule which is subscribed to the event the *condition* is evaluated and only the *action* for which the condition evaluates to true is performed. Modeling the rules therefore does not require a predefined ordering of the rules. The rule model specializes attributes from the domain model into terms, where terms are combined into a *simple-expression* of the syntax: $< leftterm, comparisonoperator, rightterm >$. Simple-expressions support the comparison-operators: $=, \neq >, <, \geq, \leq$ and can be combined into a *complex-expression* of the syntax: $< simpleexpr, logicaloperator, simpleexpr >$. The complex expression support the logical operators: conjunction (AND), disjunction (OR) and negation (NOT).

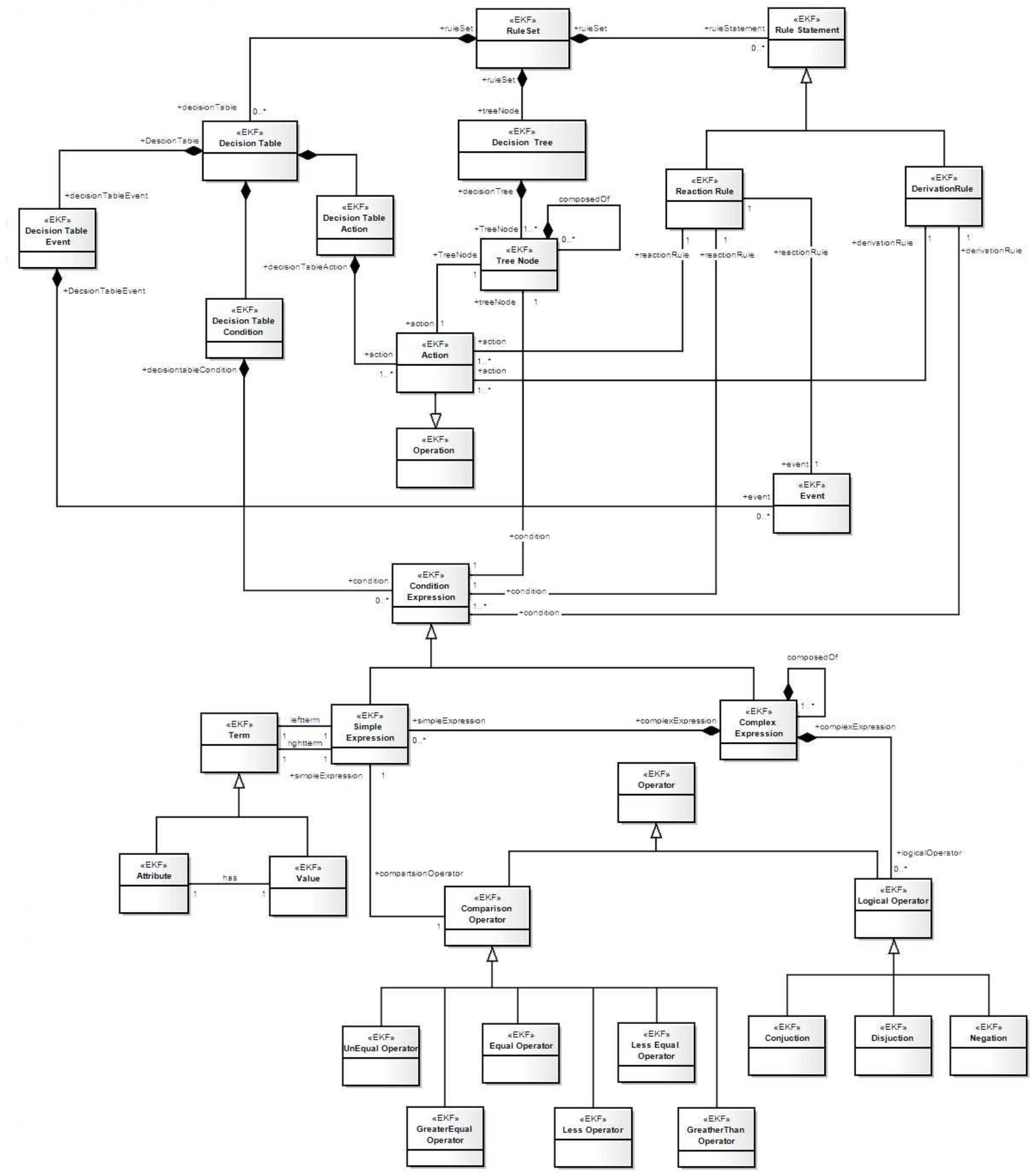


Figure 3.4: Meta-model of the EKF rule language.

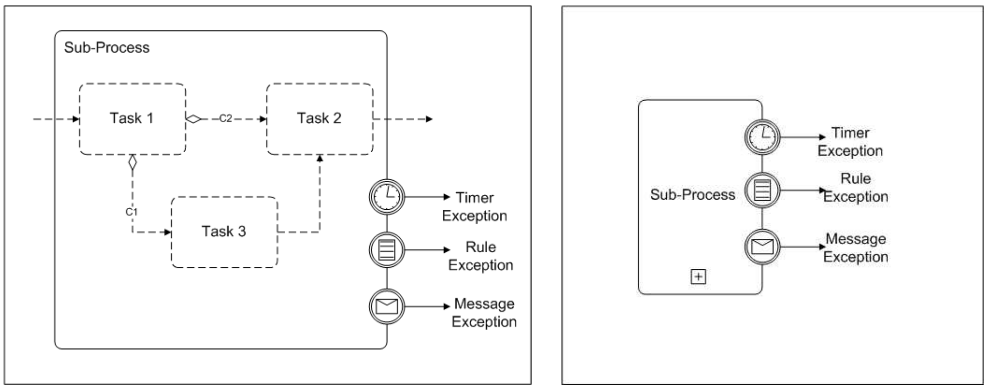
#### 3.1.2.3 Process Modeling

The process model specifies the flow of tasks to constrain the order in which tasks need to be performed. The goal of the process is to define a number of tasks that separates the control of work, such that each task is performed by a certain actor assigned to a certain role in the process. Choices in the process can be constrained by rules, such that the decisions influence the continuation of the control flow. A process model describes what the business does and why this is done, but should not say how it is done in a specific organization. In this way the process reflects the goal from sale to delivery an organization maintains, which is part of the supply-chain throughout and across the organizational boundaries. Examples of an EKF process models (mortgage process) is included in the case study (see Appendix D).

The EKF process languages defines a Business Process Diagram (BPD), which is based on a flowcharting technique tailored for creating graphical models of business process operations. Everest adopted the graphical objects of the Business Process Modeling Notation (BPMN), but have defined Everest specific semantics when constructing models. As the EKF process language does not complement the semantics of BPMN, we refer to the BPD as EKF process language rather than BPMN. The EKF process models are used for the communication of the process design among the stakeholders and forms the basis for the actual implementation of the process in terms of domain and rule models. The EKF process language bridges the gap of the business process design and process implementation in the Everest modeling approach.

A BPD is a network of graphical objects, which are composed in such a way that they constraining the order in which the task must be performed. A BPD is made up of a set of graphical elements, which promote the development of simple diagrams. These elements where chosen to be distinguishable from each other and to utilize shapes familiar to the Everest business engineers. A BPD has a small set of core elements, so that modelers do not have to learn and recognize a large number of different shapes. The elements and relations between these elements are referred to as the syntax of the EKF process language. The following elements are supported by the EKF process language: *process*, *sub-process*, *task*, *event*, *flow* and *decision-point*.

The *process* and *sub-process* are semantical similar elements, the main difference between them is that a process does not allow exception handling. The *sub-process* groups one or more tasks into a hierarchical structure, which together aim to accomplish a certain goal in the process. For the sub-process two representations can be identified: *expanded sub-process* (see Figure 3.5(a)) for which the internal tasks of the sub-process are exposed or *collapsed sub-process* (see Figure 3.5(b)) for which the internal processing of the sub-process is contained by the sub-process. The sub-process allows exception handling, such that the exceptions affect all task contained by the sub-process. Three types of exceptions are supported: timer exceptions, rule exceptions and message exceptions. The *timer exceptions* are caused by the expiration of a timer constant; the *rule exceptions* are caused by the violation of a rule; and the *message exceptions* are triggered by external systems.



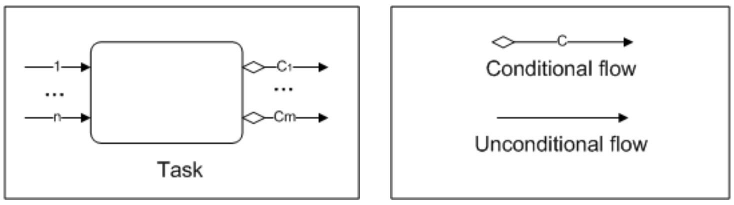
(a) Expanded Sub-process (b) Collapsed Sub-process

FIGURE 3.5: Sub-process in the EKF process.

The internal processing of the *task* is performed at a more fined-grained level in the EKF modeling dimensions, such that the processing of the task itself is not part of the process model. In the EKF the actual processing of the tasks is performed through channels of the connectivity layer either as automated or manual tasks[3]. In this thesis we consider the task as an atomic unit of work. A more specific definition: *'the set of actions a human or machine must undertake to accomplish a goal in the process'*. In the EKF process language a task is presented as a rounded rectangle, which allows multiple incoming flows triggering the activation of the task and multiple exclusive outgoing flows enabling the proceeding of the task after its completion (see Figure 3.6(a)). Actors are assigned to roles to indicate which individual groups are authorized and responsible to perform a certain task.

The *flow* specifies the order in which the tasks or sub-process must be performed or events must be processed. A flow is represented by a solid line with a solid arrowhead, constraining the order in which the tasks can be performed in the process or specify at what point in the process events must be processed (see Figure 3.6(b)). Two types of flows are supported in the EKF process language:

[3]The connectivity layer specialized various channel types allowing automated task by means of a rule model and manual task by means of a combination of the interaction and presentation model.



(a) Task (b) Flow

FIGURE 3.6: Task & flow in the EKF process.

- The *conditional-flow* requires to satisfy a certain condition expression, before the flow can be preceded.
- The *unconditional-flow* directly results in continuation of the process.

The *event* controls the temporal semantics of the process, by causing a state change on the occurrence of events. An event is represented by a circle and is something that happens during the course of a business process. These events affect the flow of the process and have a cause (trigger) and/or impact (result). In the EKF process language the following classification of events are defined: *start point events*, specify only the cause; *intermediate event*, specifies both the cause and impact; and *end point events*, specifies only the impact.

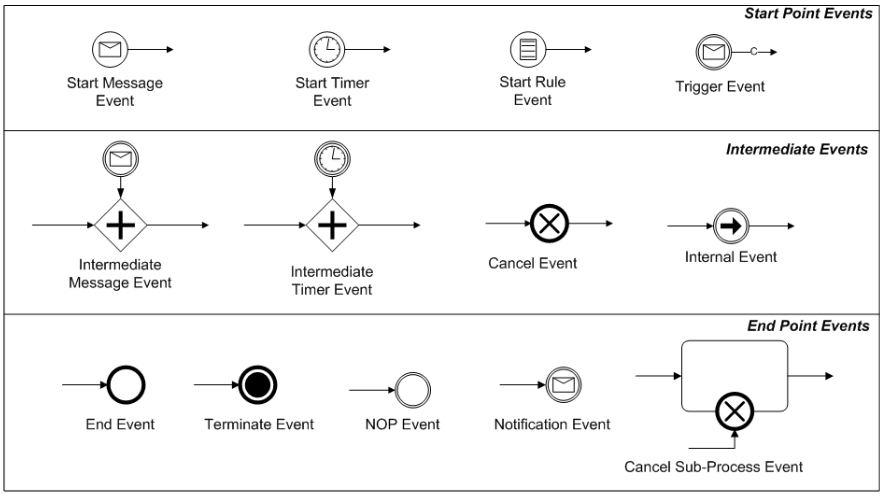


FIGURE 3.7: Events in the EKF process.

The following event types are supported by the EKF process language (see Figure 3.7):

- The *start events* activates the process and results in the creating a new instance in the process. The following start event types are supported: message event, timer event or rule event. The timer events waits for the expiration of a timer constant; the message event waits until a certain message has been received; and rule event acts on the violation of a rule.

- The *trigger events* is an external event which causes all task instances in the a sub-process to be terminated and starts processing from the event forward, but only if the condition-expression is satisfied.
- The *intermediate events* are the events, which need to be received before the process can proceed. Various types of intermediate events are supported: timer events, message events and rule events.
- The *cancel events* result in the termination of all task instances in the (sub-) process and creates a new instance to proceed processing from the event forward. Cancellation events are triggered as an impact, generally after the completion of a task.
- The *internal events* are used to notify the process engine that a certain task has been completed. Unlike intermediate events the internal events do not interact with external systems, but are used for EKF internal purposes only.
- The *end events* is the final event resulting in the completion of the process.
- The *terminate events* is a final event resulting in the termination of the entire process.
- The *NOP events* is a final event for which no additional actions are undertaken.
- The *notification events* are the events, which notify external actors or processes of the occurrence of a certain event. The notification event result in actions, which are accomplished through either system-to-system or system-to-human interaction.
- The *cancel sub-process events* allow the cancellation of a sub-process and terminates the processing of all instances of tasks contained by the sub-process. Occurrence of this event directly results end of the sub-process.

The EKF process language specifies a *decision-point* for controlling the complex decisions in the process. The decision-points in the EKF are implemented in terms of process decision tables, for which conditions are evaluated to derive conclusions about the continuation of the process.

The following types of decision-points are supported by the EKF language (see Figure 3.8):

- The *OR-split-decision-point* is a point in the process where the process can proceed processing one or more branches based on the evaluation of a set of non-exclusive conditions. Each branch for which the condition evaluates to true is considered as

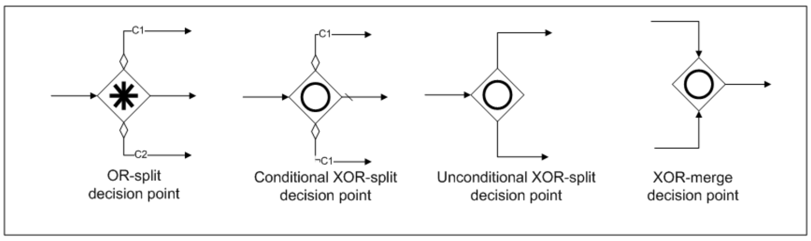

C1
C2
OR-split decision point
C1
¬C1
Conditional XOR-split decision point
Unconditional XOR-split decision point
XOR-merge decision point

FIGURE 3.8: Decision-points in the EKF process.

the preceding of the process. Notice that the OR-split-decision-point also allows unconditional branches, which are taken under all circumstances.

- The *Conditional XOR-split-decision-point* is a point in the process for which the process can choose one branch of several branches through exclusive conditions. Only the branch for which the condition evaluates to true is taken. Important is to notice that the XOR-split can also be performed directly on a task, as multiple exclusive outgoing flows.
- The *Unconditional XOR-split-decision-point* is a point in the process where the branch is not selected based on a data based decision, but based on the reception of events. Only the branch for which the first event occurs is preceded, such that the other branches are discarded when the first event has been received.
- The *XOR-merge-decision-point*, is a point in the process for which multiple exclusive branches are merged into a single branch. Important is to notice that a XOR-merge-decision-point can also be performed directly (as multiple incoming flows) on a task.

In this section we provided a more detailed description of the syntax of the EKF process language, which is an important result as no such description is available to us at the start of this project. Still we need to asses the semantics of the EKF process language as the EKF process language does not directly satisfy a certain computational property. Therefore we can classify the process model as a CIM model at the M2 level of MDA. From the observation of the elements and their relations we derived the syntax of the EKF process language and presented it as an UML meta-model (see Figure 3.9).

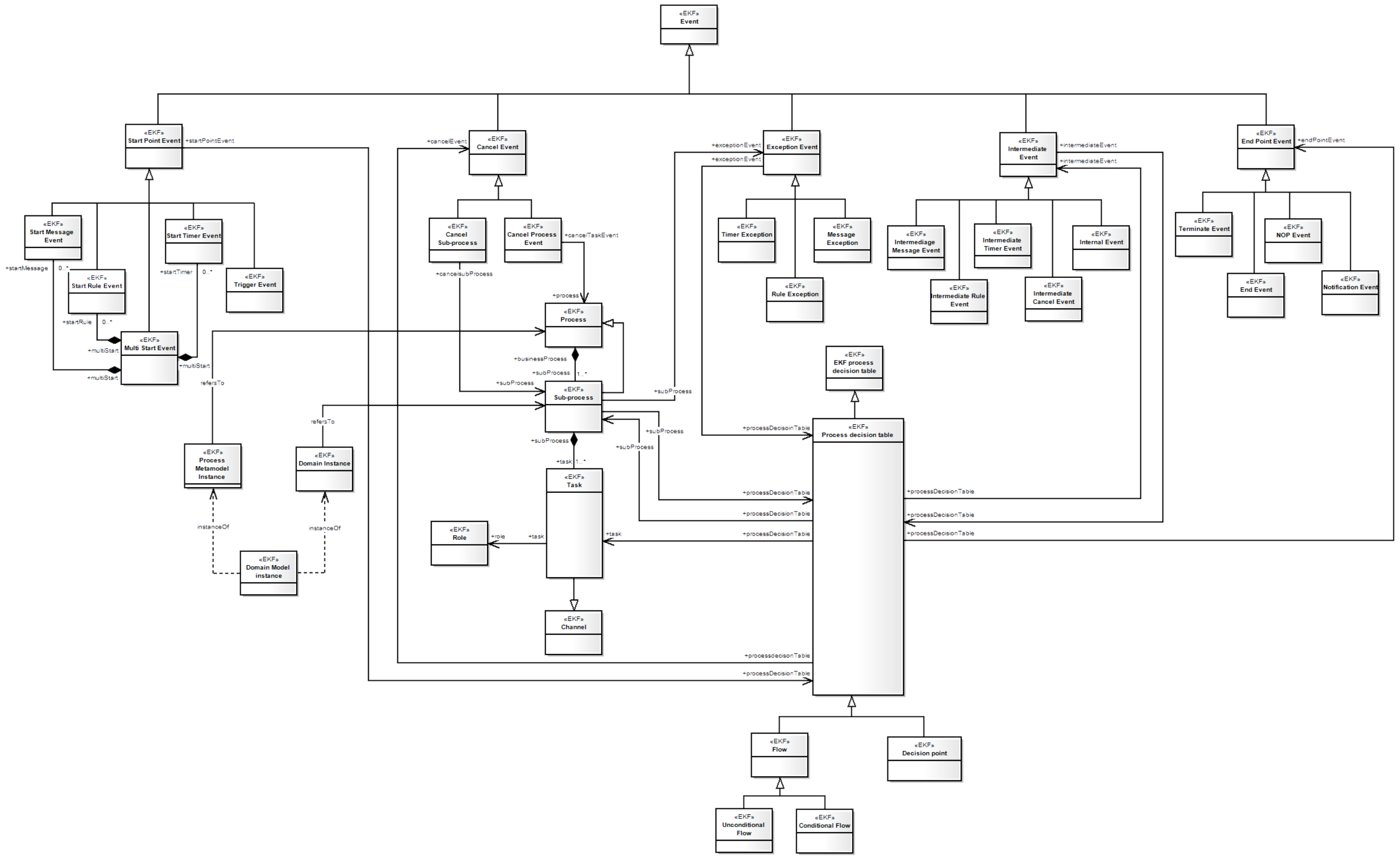


FIGURE 3.9: Meta-model of the EKF process language.

#### 3.1.2.4 Connectivity Layer

The connectivity layer[4] provides the channels that can be used for the interaction and the synchronization with external systems and tools. A more specific definition: *'an interoperable building bock for which the semantics are specified by the underlying models or programming logic'*. The connectivity layer operates independent of the channel implementation (implemented as a generalization of extensible channel types) (see Figure 3.10). The channels are therefore strongly related to the services of the enterprise architecture. The EKF channel is specialized into different channel operations with each input and output variables. The channel types are divided by its semantical properties: transformation channel (transformation from one data format into another), decision channel (derive data from the domain model by applying a set of rules), data channel (update and retrieve data from the domain model), integration channel (programming logic). Notice that the connectivity layer is based on a predefined set of services, implemented as programming logic. The construction of the channels require technical insights and considerations, such that software engineers are still needed for the construction of these channels.

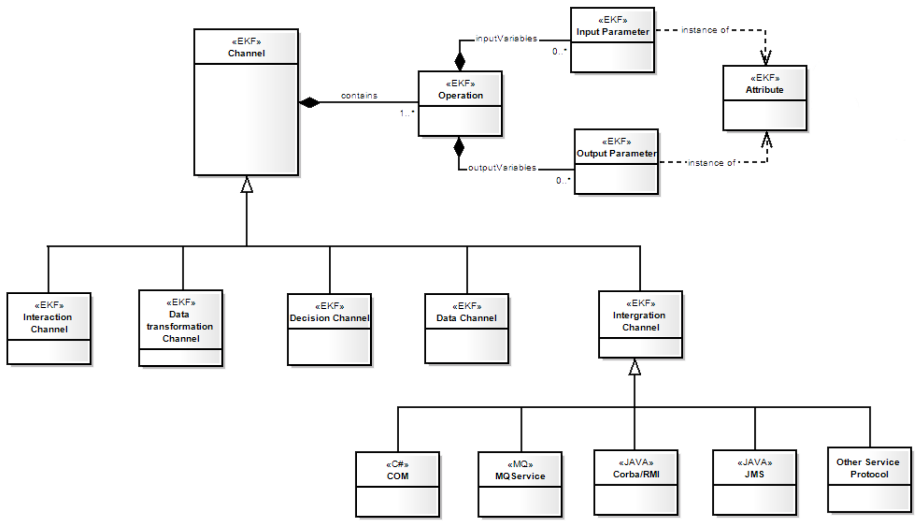


FIGURE 3.10: UML class diagram of the connectivity layer.

In the previous sections we have provided more insight into the EKF core modeling languages and defined the syntax in terms of UML meta-models and UML class diagram. This is considered as an important result for Everest, as at the start of this project no such information was present. The insights of the EKF process language syntax are of interest in the preceding of this thesis, as we aim at transforming EKF process models

[4]The connectivity layer is referred to as a layer rather than a model, because channels are deployed as programming logic, not as executable models.

(models constructed with the EKF process language) into BPEL code. In the remainder of this chapter we focus on the way EKF process models are deployed on the EKF, to get more insight into how the EKF process models are implemented and enacted in the EKF. Enactment is important in the sense, that they control the run-time semantic of the EKF process models.

## 3.2 How are the process models deployed on the EKF?

In the previous section we have focused on mainly the syntactical perspective of the process modeling approach of Everest. In this section we give a more detailed study of how the EKF process models are implemented and enacted in the EKF.

### 3.2.1 How is the process model implemented in the EKF?

In this section we give an overview of how the process models are actually implemented in terms of the EKF domain and rule models. The structural aspects of the process, described by the meta-model, is implemented in terms of entities, attributes and their structural relations of the domain language. The state transition in the process (decision-points and flows) are defined as a specialized decision tables (process decision table), which directly inherit the properties of the decision table of the rule model. This *process decision table* specifies the performing of actions through operations, triggering of an events or activation of a tasks or sub-processes, based on the occurrence of certain events and in case certain condition-expressions are satisfied.

The flows and decision-points are modeled explicitly, but are at deployment concealed by the way the process decision table are implemented. Table 3.1 specifies the grammar of the EKF process decision table for which the following rows are considered:

- The sub-process is optional and defines to which sub-process the decision point or flow belongs.
- The event guards the preceding of the flow by waiting for the occurrence of a certain event.
- The alternative events are optional and specify the second event guarding the preceding of the flow.
- The condition, guards the preceding of the flow through the evaluation of a condition-expression.

- The action specifies the activation of: sub-processes, tasks, events or operations (composed of input ($\beta$) and ($\alpha$) output variables). Notice that multiple alternative actions are optional.

| EKF Decision Table | |
|---|---|
| Sub-process (optional) | $SubProcess\|*\|?\|[]$ |
| Event | $Event\|*\|?\|[]$ |
| Alternative Event (optional) | $Event\|*\|?\|[]$ |
| Condition (optional) | $ConditionExpression\|*\|?\|[]$ |
| Alternative Condition (optional) | $ConditionExpression\|*\|?\|[]$ |
| Action | $SubProcess\|Task\|Event\|Assign(\alpha,\beta)$ |
| Alternative Action (multiple optional) | $SubProcess\|Task\|Event\|Operation(\alpha,\beta)$ |

TABLE 3.1: The syntax of the EKF process decision table

### 3.2.2 How is the processes enacted on the EKF business engines?

The EKF process is controlled by interaction between the: portal engine, event manager, process engine, rule engine and ESB. The portal engine enables the interaction with the human actor and allows the request and retrieval of the workitems through the worklist manger. The process engine is the central controller of the process and is responsible for maintaining the process instances and acts on the events from the event manager. The domain storage implements the structural aspects of the EKF process language (meta-meta-model) and maintains the data in the process in terms of a domain model (meta-model). The rule engine is responsible for the decision taking in the process, by deriving conclusions from the domain model (see Figure 3.11). A more detailed description of the process and rule engine is provided, as they control the most considerable part of the actual process semantics.

The process engine maintains instances for a certain process model also referred to as a case. A *case* is an independent instance that maintains the results of the (sub-)processes, when routed through the process. The semantics of the routing of the cases through the processes is presented in the UML state transition diagram of figure 3.12(a), for which the following states are identified:

- *Initialized*, the domain storage has initialized the process syntax model and created an instance to the domain data of this model for a specific case. Also the domain model which is responsible to maintain the process data is initialized, for which the instance is maintained by the process engine.

- *Running*, there still exist one or more active (sub-)processes mainainting active tasks.

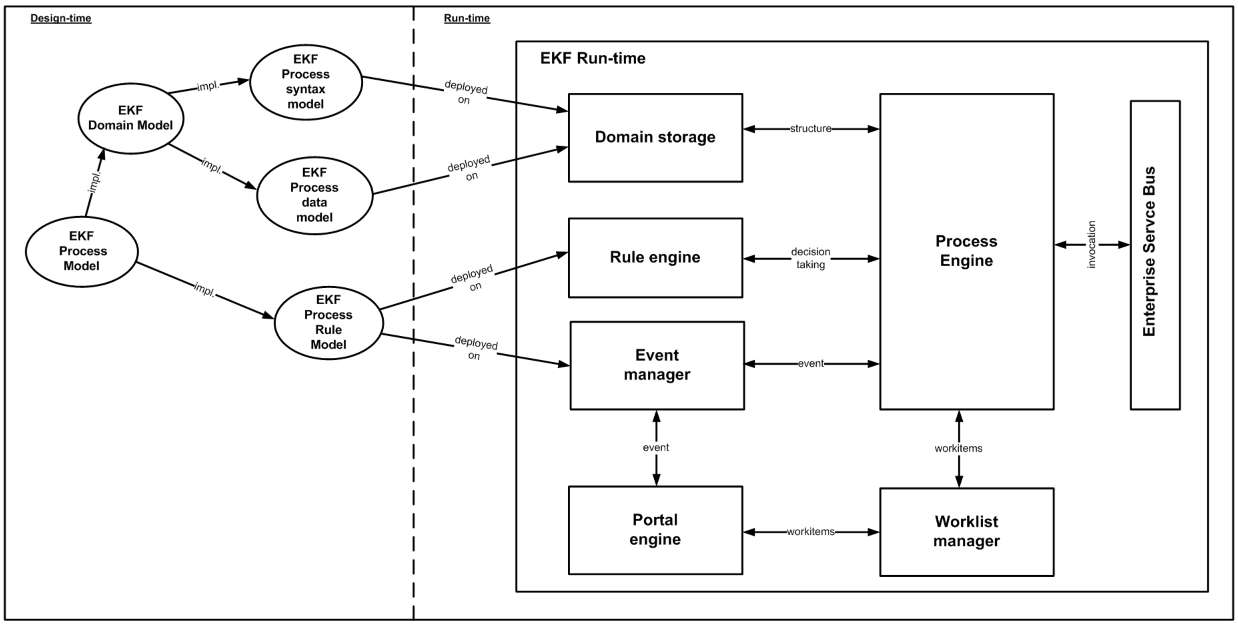
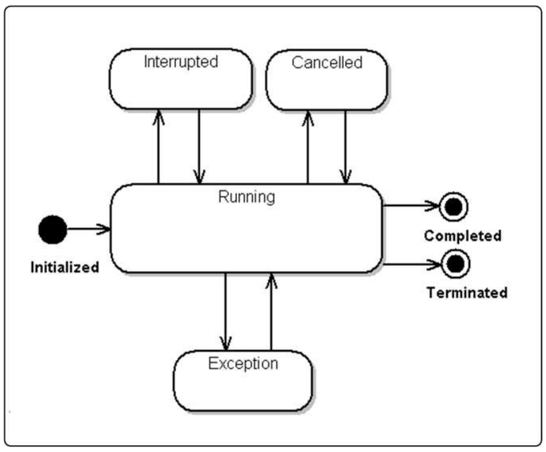


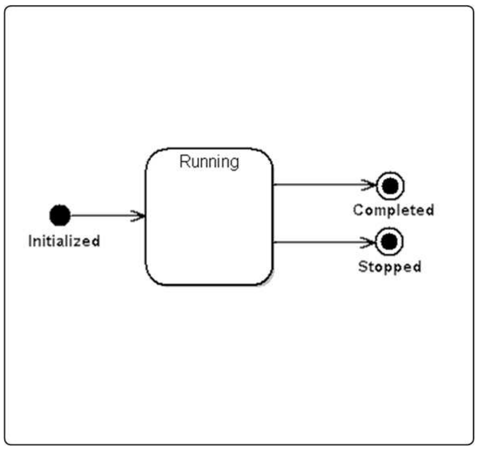
FIGURE 3.11: Deployment of EKF process models in the EKF run-time(impl.=implements).

- *Completed*, the process instance is completed, in case no more active sub-processes exist in the process.
- *Exception*, the case has received an exception, such that the rule engine is requested to update the state of the process. The active instances which belong to the process are not affected by the exception state. After the state is updated the process determines if new instances must be created in order to handle the exception.
- *Terminated*, the case has been terminated such that all active instances for the case are stopped.
- *Interrupted*, the case has received a satisfied trigger event, causing the all active instances for the case to be stopped. The rule engine is requested to determine the continuation of the process after the occurrence of the trigger event.
- *Canceled*, the case is canceled through the occurrence of a cancel (sub-process) event, causing all active instances for the case to be stopped. The rule engine is requested to determine the continuation of the process to handle the cancellation.

The goal of each task is mostly related to the input of information, also referred to as knowledge that becomes available when a task has been completed. As the case is routed through the process, the information that has become available in one task is passed though the next task. The cases in the process are controlled through variables derived from the domain model, which abstract the goals that are accomplished throughout the



(a) State Transition Diagram sub-process



(b) State Transition Diagram task

FIGURE 3.12: State Transition Diagram EKF process.

process (the conditions that have been met in the process). The derivation and assignment of this variable is the responsibility of the rule engine. The EKF process engine has a considerable limitation, as only a single variable is supported in a (sub-)process. Therefore it is not allowed to use multiple instances of a same task in the process[5]. The semantics of the task are controlled on a lower-level of abstraction compared to the (sub-)process. The semantics of the task is therefore represented as an independent UML state transition diagram (see Figure 3.12(b)), for which the following states are identified:

- *Initialized*, the domain model is initialized by the domain storage and an instance to the domain data is received by the process engine.
- *Running*, a task is being performed and the process engine waits until the task is either completed stopped or accepted.
- *Completed*, the task has satisfied its post-condition, resulting in the rule engine to update its domain data. Based on the result of the rule engine: a new task is activated in the (sub-)process and its state is either set to running or stopped (no preceding actions for that task are required to be performed).
- *Stopped*, a (sub-)process has explicitly been aborted, such that all tasks contained by this (sub-)process are aborted.

The domain storage is responsible to initialize the models and maintain the integrity and consistency of the factual knowledge in terms of domain data. The rule engine plays an important role in the state transition when the cases are routed through the process.

[5]Multiple instances are supported by the application flow at a more fine-grained level of the EKF, which is not part of the process dimension in the EKF.

After the completion of each task the data variables governing the state of the process are updated, by deriving which goals in the process have been satisfied. The rule engine uses backwards chaining inference mechanism to derive if these goals goals (described by rules) have been met. Each decision in the process requires the rule engine to update the state of the process variables.

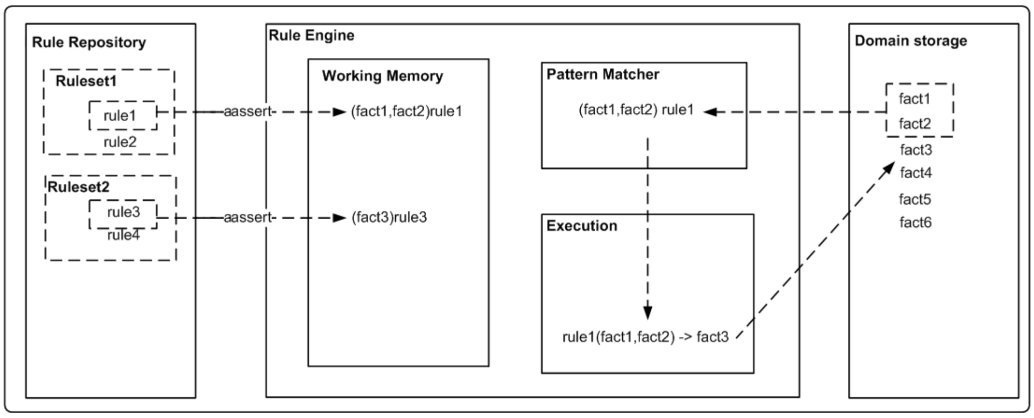


FIGURE 3.13: Logical components in the rule engine.

To accomplish the inference of the domain data the rule engine is composed of mainly three general components as presented in figure 3.13:

- The *rule repository* contains the heuristic knowledge. The heuristic knowledge is the knowledge, which can be derived by judgment of the facts in the data repository. The EKF mainly supports decision tables, decision trees and rule statements to model rulebases in the rule repository.
- The *rule engine* controls the execution of the rules. The primary purpose of the rule engine is to apply rules when the set of facts (domain data) is complete. The *working memory* holds both the premises and conclusion of the rules, such that the derivation of a conclusion can trigger the next rule to be fired. At run-time the working memory allows the following actions on rules: *assert*, adds rules to the working memory; *retract*, discards active rule from the working memory; *modify*, updates and active rule in the working memory.
- The *pattern matcher* is responsible matches the facts with the rules and determine if they are eligible to fire. The executable rules are eligible to fire if the facts fulfill the required rules to derive its conclusions.

The ESB provides the actual channels through which the task is completed. The semantics of the task is defined by the way the channel is implemented in the connectivity layer. Each channel is governed with input and output variables, which represent the information flow through the task. In the preceding of this thesis we assume that channels are performed synchronous, which implies that the task is performed as an atomic channel invocation.

In this chapter we have presented the enterprise architecture of the EKF and how Everest adopted MDA as a strategy for modeling. We presented the syntax of the EKF process, which is considered to be an important result for Everest as no such description was available to us a the start of this project. We also presented a more detailed description of how the EKF process is deployed in the EKF, to give more insight into the way Everest implements the process models. The deployment is important as the semantics of the EKF process are subsistent to the way the EKF process are enacted. In the following chapter we continue with a more detailed study of BPEL, as Everest aims at adopting this language to improve their deployment strategy in the EKF process available.

Chapter 4

# Business Process Execution Languages for Web Services

In this chapter we aim at answering the question: *'What are the key features and syntax of BPEL?'*. To answer this question we start with a general introduction of BPEL (see Section 4.1) and overview of Web service technology (see Section 4.2), this is followed by a more detailed overview of the syntax[1] and semantics of BPEL (see Sections 4.3, 4.4 and 4.5).

## 4.1 Introduction to BPEL

The Business Process Execution Language for Web Service (BPEL4WS or BPEL) was introduced by (IBM, BEA and SAP, 2002) as a defacto standard for describing the behavior of Web services in terms of process models at different levels of abstraction. BPEL has emerged as a compromise between the Web Service Flow Language (WSFL) [Leymann, 2001] and XLANG [Thatte, 2001]. BPEL therefore supports both block-oriented language features from XLANG and graph-oriented features from WSFL. The first version of BPEL (1.0) was published in August 2002 [IBM et al., 2002], the second version (1.1) [BEA et al., 2003] in May 2003 and the latest version (2.0) [OASIS, 2006] is at the time of writing in working draft. BPEL has been adopted by the industry as the leading process interchange and enactment standard.

BPEL allows the business to keep the internal business protocols separate from the cross-enterprise protocols. Separation of the private from the public protocols is important for two reasons: first, the business does not want to reveal its internal processing (e.g.

[1]We used the ideas of [Akehurst, 2004] for formalizing the syntax of BPEL in terms of meta-models

data management and decision taking) to their business partners; second, the private protocols need to be changed without affecting the public protocols. BPEL defines the coordination of multiple service interaction with the partners from a single-point and can therefore be classified as orchestration language.

The BPEL process definitions can either be a fully executable model (executable BPEL) or abstracted model (abstract BPEL). Abstract BPEL aims at defining the role of the protocol, without considering the details of the process implementation (typical PIM). Executable BPEL specifies the behavior of the process at a more fine-grained level, in terms of the composition of Web services. The executable BPEL definition can directly be executed by a BPEL engine (typical PSM). In this thesis we are mainly interested in BPEL, as Everest main objective is to adopt a BPEL engine allowing the enactment of executable BPEL in the EKF. BPEL provides a grammar based on eXtensible Markup Language (XML), similar to a programming language, but is less complex and specifically suited for defining business processes. BPEL is an extension of Web services and build upon Web Service Description Language (WSDL) of the Web service stack to define the interactions between the partners. The interaction with each partner occurs through Web service interfaces, where the structure of the relationship at the interface level is called a partner link. As Web services are an important aspect of BPEL we first give an overview of Web service technology.

## 4.2 An overview of Web services technology

In this section we give an overview of the Web service stack, such that readers which are already familiar with Web services can proceed reading the next section. The core Web service stack (see Figure 4.1) consists of standards to accomplish interoperable platform independent and loosely coupled services by means of Web services. BPEL is build on top of the Web service stack, such that Web service are the backbone of BPEL.

XML[2] is used as an generic data interchange format, through XML documents. XML Path (XPath)[3] and XQuery are languages for addressing portions of an XML document (query) or computing values based on the content of an XML document.

The WSDL[4] is a specification used for defining how to describe Web services. WSDL enables clients to locate a Web service and invoke any of its publicly available functions. The Simple Access Protocol (SOAP)[5] is an XML based communication protocol for

[2]see: http://www.w3.org/XML
[3]see: http://www.w3.org/TR/xpath
[4]see:http://www.w3.org/TR/wsdl
[5]see: http://www.w3.org/TR/soap

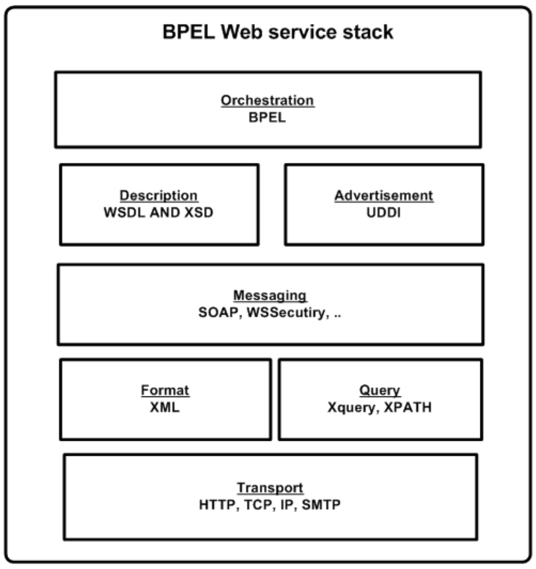


Figure 4.1: The Web service stack.

applications to access Web services on any platform. The XML Schema Definition Language (XSD)[6] is used to specify the layout of an XML documents, which are provided as input or output of Web services. The eXtensible Stylesheet Language Transformation (XSLT) is a language enabling the transformation of one XML document format into another XML document format.

The Universal Description Discovery and Integration (UDDI)[7] prescribes a set of standard interfaces for accessing information about Web Service location. UDDI is an industry effort to enable dynamic discovery for Web services.

BPEL is ans extension of the Web service technologies, specifically designed for defining business processes characteristics by means of: partner links, base and structured activities, data, transaction, compensation, exception and event handling. In the following sections we present the syntax and semantics, by providing a more detailed overview of the key features.

## 4.3 Process and partner links

The top level element in BPEL is the *process* (see Figure 4.2), which contains one or more subsequent elements. The process is a specialization of the structured *scope activity*, which allows the process to be decomposed onto activities and maintaining variables, transaction, compensation and exception handling. The specialization from

[6]see:.http://www.w3.org/XML/Schema
[7]see: http://www.oasis-open.org/committees/uddi-spec/

the scope activity allows the nesting of activities, such that a hierarchy of activities can be created as activities can contain activities. Every process is triggered by a receive activity and ends with an invoke or reply activity. BPEL uses the concept of invocation of Web services for the interaction by means of partner activities.

The *partner links* allow BPEL to interact with Web services in the process. The partner link is assigned to a (WSDL) port type, which is uniquely identified through its name. Each partner link is characterized by a *partner link type* defined in a WSDL definition. For a partner link two type or roles can be set: a producer and consumer role. A partner link with only a consumer role does not need to know about its callers (typical a-synchronous). A partner link assigning both producer and consumer roles specifies a typical synchronous interaction. In BPEL these interactions are accomplished through the partner activities: receive, reply and invoke. These activities enable the use of different type of partner interactions.

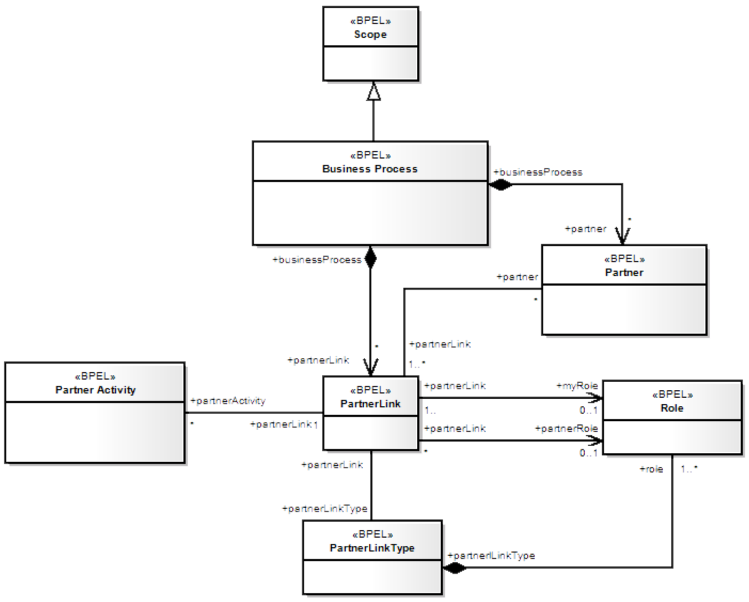


FIGURE 4.2: Meta-model of BPEL process.

## 4.4 Activities

The *activities* (see Figure 4.3) correspond to the tasks that have to be performed in order to complete the process. The values, variables and expressions are considered to determine the routing of the cases through the process. BPEL recognizes two types of activities: base and structured activities. The *base activities* correspond to atomic actions, which cannot contain other activities: interact with a service, manipulate exchanged data or handle exceptions. The base activities have attributes and elements that can be used to specify certain properties. The base activities allow modeling the control flow, data and fault handling in the process. The *structured activities* allow nesting of the activities, used to compose the order in which the base activities need to be performed.

BPEL contains the following eight partner activities: *receive*, wait for message from external partner; *invoke*, represents the synchronous operation of an external partner; *reply*, an a-synchronous result to an external partner; *wait*, pauses for a certain period of time; *assign*, copies data from one place to another; *throw*, indicate errors during the process execution; *compensate*, rolls back transactions when errors occur; *empty*, do nothing. Three partner activities are responsible for the invocation of Web services in BPEL: invoke, receive and reply. For an invoke activity the partner link, service operation and port type need to be specified to invoke services of the business partners. The invocation can either be synchronous (request/reply) or a-synchronous (one-way). BPEL provides services to its partners through the receive and reply activities. The receive activity blocks the activity until the corresponding message arrives, while the reply activity is used to send a request from a previously accepted receive activity. The receive activity can also be used to initiate the process instance by setting the value of the property *createInstance* to *yes*.

BPEL identifies the following six structured activities: sequence, switch, pick, while, flow and scope. The *sequence* activity contains one or more activities that have to be performed sequentially in a lexical order. The *switch* activity supports the conditional routing of activities. It contains the ordered list of one or more conditional branches defined by case elements, only the first branch for which the condition evaluates to true is considered. The *otherwise* branch can be defined as the default branch to make sure that there is an alternative path that can be taken, in case none of the case conditions are satisfied. The switch activity is completed when the activity of the chosen branch has been completed. The *pick* activity captures the race conditions based on timing or external triggers. Like the switch the pick has a set of branches, but instead of conditions it defines a number of events that need to be occurred before the corresponding activity can be initiated. Two types of events are provided: message events (*onMessage*), waits for occurrence of external event and alarm events; timer events (*onAlarm*), waits for the expiration of a timer constant. The pick activity acts only on the occurrence of the first event, such that the occurrence of the first events discards the occurrence of the events of the other branches. The *while* activity supports repeated behavior of activities. The while activity is performed until the (pre- or -post) condition no longer evaluates to true. The *flow* activity provides the parallel execution and synchronization of activities.

BPEL support control links to extend the control of the dependencies between activities nested within the flow activity. The *scope* activity is used to group activities into blocks, where each block is treated as a unit for which the same *events*, *compensation* and *fault* handling is applicable. This means that the handlers are only visible from within the boundaries of the scope activity. Because business processes are long running, it is infeasible to keep open its transaction, therefore BPEL provides compensation handlers rather than a mechanism to rollback transactions. The compensation handler is executed using a *compensation* activity, which is explicitly triggered by fault handlers. The BPEL engine is responsible to throw faults during the execution of the activities within a scope. The generated faults can be handled in the BPEL code using a catch construct. The event handler enables the scope to react to events or expiration of timers at any point during the processing of the scope.

The structured activities: sequence, flow, switch, pick and while offer a mechanism to model features where one activity cannot start until one or more activities on which they depend are completed. In BPEL this concept is called control links, which can be referred as a conditional transition between two activities. Two types of control links are provided by BPEL: join-condition and transition-condition. A *join-condition* is associated to an activity, which expresses a condition that defines the dependency between two activities. The conditional dependency defines the required synchronization between activities during parallel execution. The *transition-condition* defines a condition, that specifies when activities are enabled or disabled for execution.

## 4.5 Data Handling

Most processes need to maintain application data during the course of their execution. The data is initialized when the process starts and is subsequently read and modified during processing. BPEL supports defining a set of variables, which can be passed through Web services as input and output variables in terms of XML documents. Variables in BPEL can be set in various ways: bound variable to an inbound activity such as pick, receive or event handler; bound to the output of an invoke activity; explicitly assigned through an assign activity (see Figure 4.4(a)).

The variables are used to hold information of the message exchange between the process and its partners as well as internal data, which is private to the process. A process variable has a name that is unique to its scope and the type defined in terms of a WSDL message, composed of types from the XSD type definitions (see Figure 4.4(b)). BPEL adopted XSD to define the layout of the XML data documents, which are defined as input and output variables of the invoked WSDL operations (see Figure 4.4(c)).

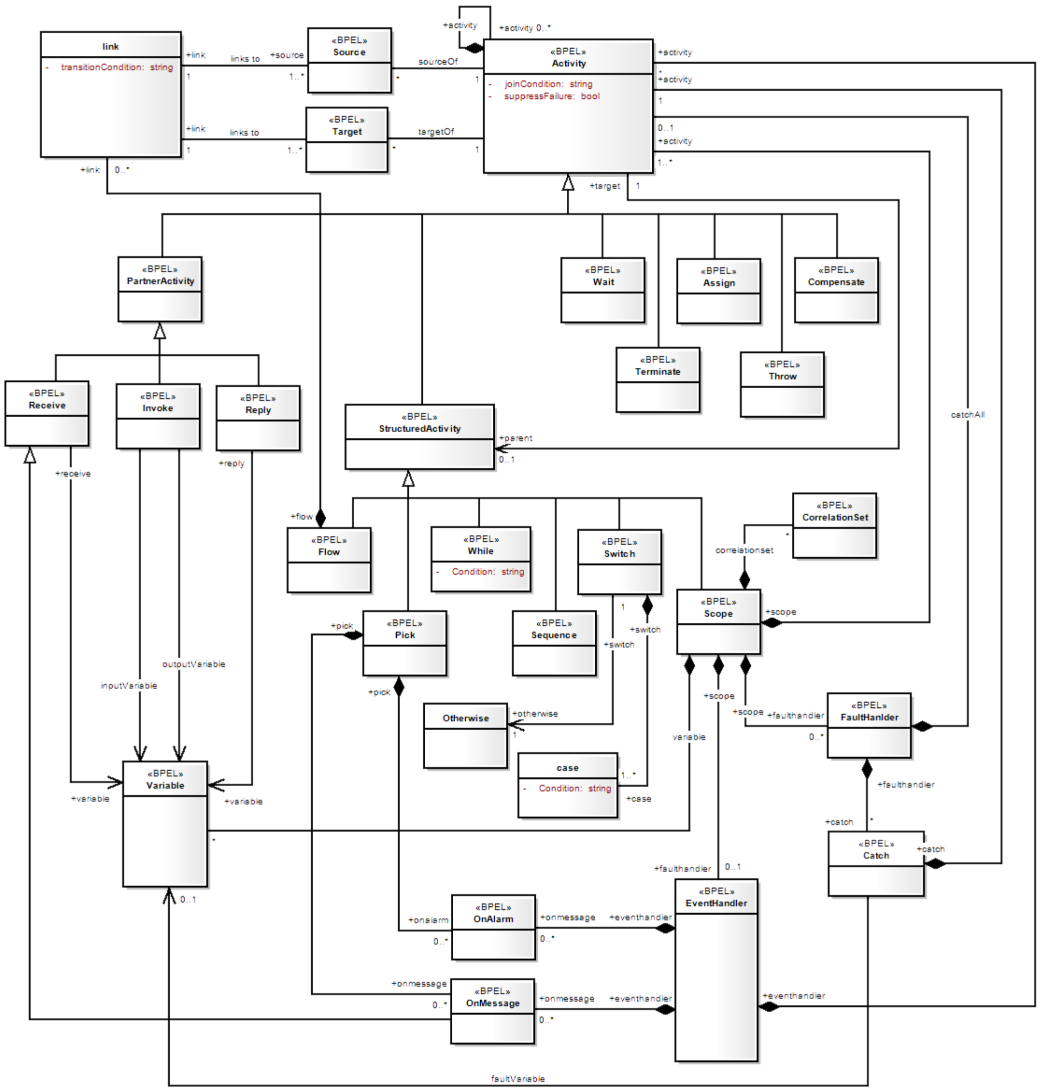
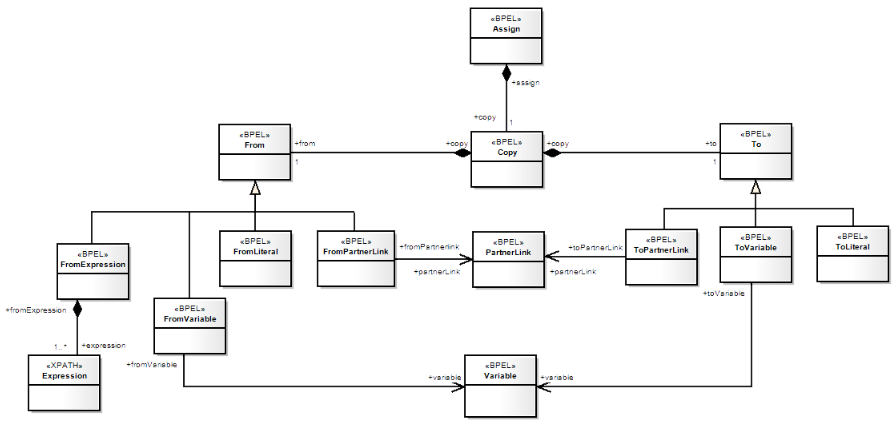
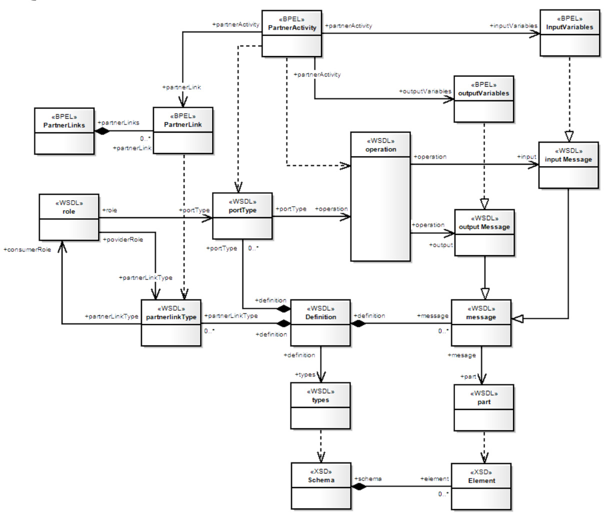


FIGURE 4.3: Meta-model of BPEL activity hierarchy.

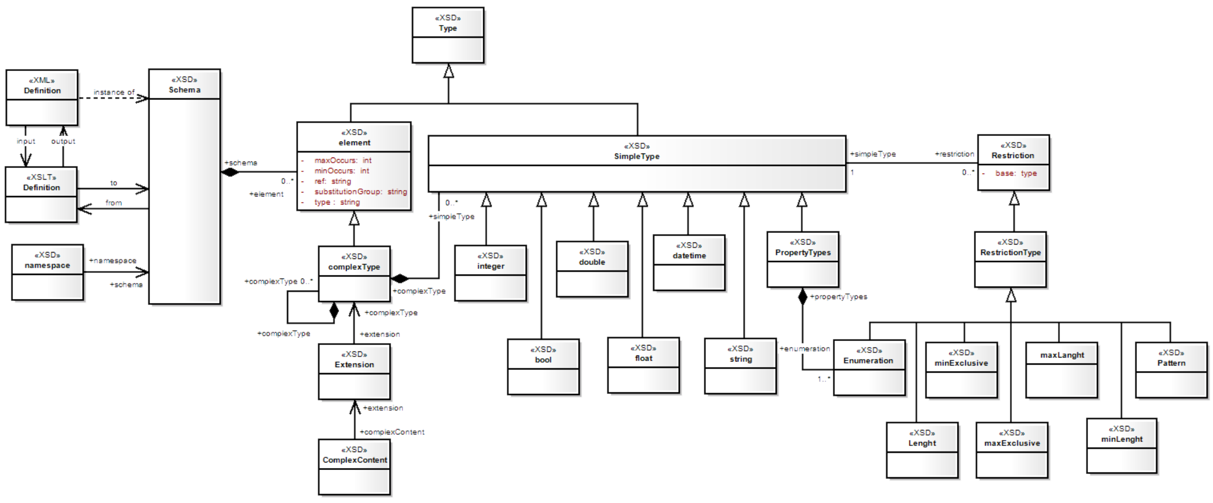
In this chapter we provided an overview into the key features of BPEL providing more insight into the syntax and semantics of BPEL. This is required in the following chapters, where we determine the language translations between EKF process language and BPEL, toward an approach to transform EKF process models into BPEL code.



(a) Meta-model of the BPEL assign



(b) Meta-model of the BPEL WSDL extensions



(c) Meta-model of the BPEL XSD extensions

Figure 4.4: Meta-models for data handling in BPEL.

# Chapter 5

# Transform the EKF process language into BPEL

In this chapter we aim at answering the question: *'What are the issues and challenges to close the gap between the EKF process language and BPEL?'*, which is considered as a partial question required to answer the research question: *'How to make the EKF more interoperable by adopting BPEL?'*. In section 5.1 we give an overview of the approach followed to close the gap between the EKF process language and BPEL, for which the results are presented in section 5.2. The results of our observations is summarized in section 5.3. In this sections we only aim at identifying the issues and challenges which must be considered when translating the EKF process models into BPEL code. In chapter 6 we use these observations to address the issues and challenges toward an approach to translate the EKF process models into BPEL code.

## 5.1 Approach to transform the EKF process language into BPEL

The aim of Everest is to adopt a more generic approach to deploy their processes, increasing the interoperability of the process in the EKF. Everest is mainly interested in the adoption of BPEL to fulfill this goal. Supporting BPEL in the EKF should result in an increase of the process interoperability across the enterprise architecture, as BPEL is supported by different tools and can be enacted by generic engines. For Everest the adoption of BPEL implies that the properties and capabilities of EKF process language must be translated (language) and deployed (code) in terms of BPEL. To accomplish this we need to consider both the semantics and syntax of the EKF process models and BPEL code. For the transformation of the EKF process models into BPEL code we are only interested in one direction of the transformation. This implies that we need to evaluate if BPEL is able to express concepts recognized in the EKF process models. Nevertheless, Everest is also interested in the limitation of the EKF process language. In extend of the forward transformation we also evaluate limitations of the EKF process language by identifying concepts which are supported by BPEL, but can not be modeled in terms of the EKF process language. These limitations directly affect the expressiveness of the EKF process language.

In the Everest approach the EKF process models are translated into the EKF domain and rule models, for which the actual semantics are defined by the a combination of the EKF process engine and rule engine (as presented in Section 3.2). The adoption of BPEL requires a different approach, where EKF process models are translated into BPEL code which are deployed onto generic BPEL run-time engine(s) (e.g. IBM, Oracle and Active BPEL, etc.) (see Figure 5.1).

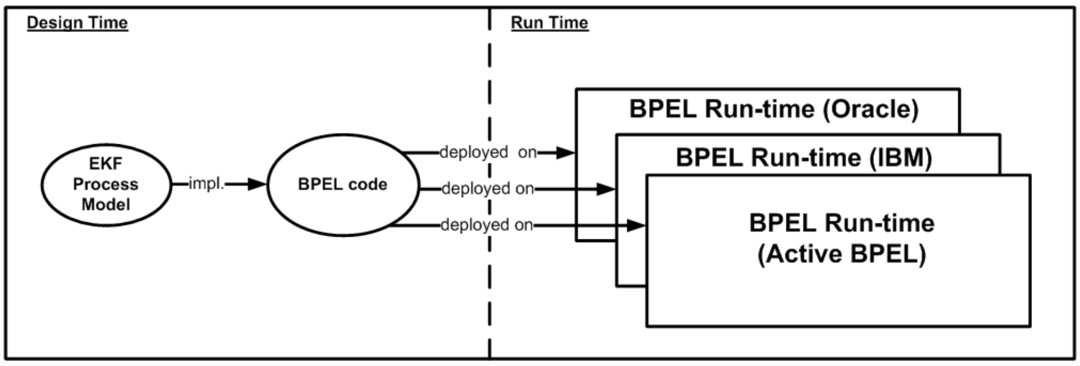


FIGURE 5.1: Deployment of EKF process models onto BPEL engine (impl.=implements).

To reduce the complexity of the model based transformation, we first consider the transformation of the EKF process language into the BPEL. According to the approach of [Bordbar and Staikopoulos, 2005] the transformation of one language into another language requires bridging the gap between both the domains and the technical spaces. The *domains* are defined as: *'the context via specific application aspects or behavior, given by programming language syntax'* and the *technical spaces* as: *'a specific working context with specific implementation technologies, tools and approaches, where applications are specified, instantiated and utilized form various tools and engines'*. To close the gap between the technical spaces and domains of the EKF process language and BPEL we need to cover both aspects by means of a domain transformation (see Section 5.2.1) and technical spaces transformations (see Section 5.2.2). In this thesis we follow the *One step refinement approach* of [Bordbar and Staikopoulos, 2005] for which a more detailed description is presented in appendix A.

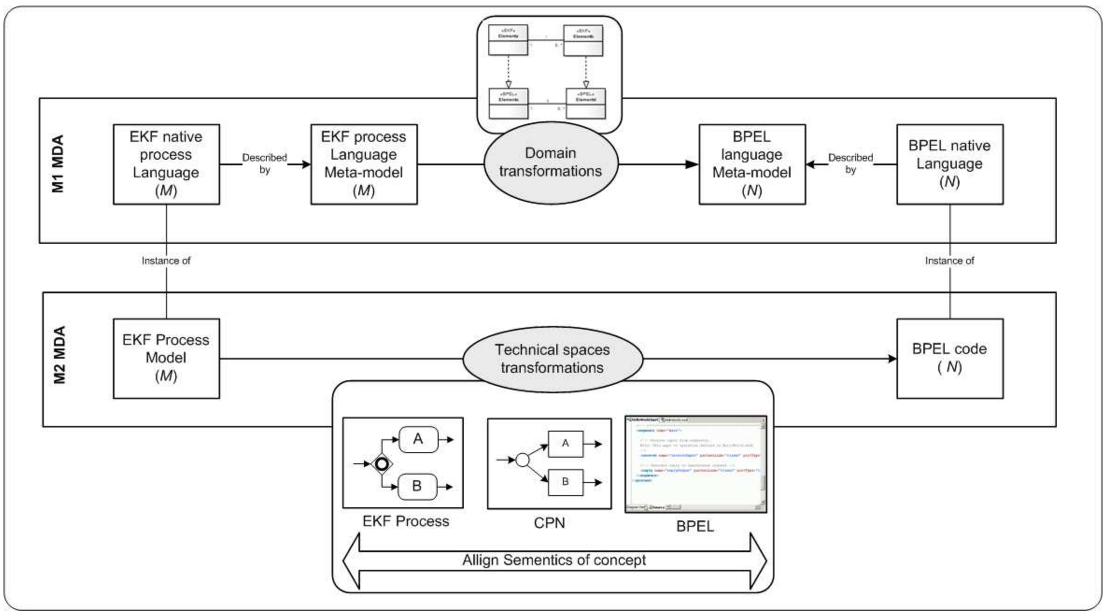


FIGURE 5.2: Transformation of EKF process language into BPEL.

The *domain transformation* focuses on closing the gap of the elements at the meta-model level of the EKF process language and BPEL. To realize such an idea, we need to establish a bridge between the EKF process language and BPEL. This is accomplished by considering each element in the meta-model of the EKF process language and determine if a corresponding (set of) element(s) exist in BPEL supporting similar semantics (see Figure 5.2). This is not an easy task, because meta-models define a complex structure for which the semantics are defined by its conceptual interpretation. Such a bridge between two meta-models is referred to as a *direct mapping*, as a solution of an element of the source language can directly be translated into one or more elements in the target language. The direct mappings should give more insight into the contextual differences between the EKF process and BPEL language at the M2 level of MDA (see Figure 5.2).

Sometimes, however it is not possible at the first attempt to close the gap of two meta-models directly through only the context. This is mainly caused by the fact that the technical spaces of languages could be rather different. Such problems often occur when one space may define or posses characteristics that the other language does not account for. A domain transformation is therefore not sufficient to close the gap between two languages entirely, in such cases the languages are considered to be different from the context. This implies that for some elements no direct mappings can be identified. This implies that one should not consider a translation directly through the context, but using concepts as a bridge between two languages. The *technical spaces transformation* should therefore address these contextual differences, by using more generic concepts supported by both the source and target languages. The aim of the technical spaces transformation is to find a solution for a predefined set of patterns for both the EKF process model and BPEL code. In this way the pattern is used an intermediate language concept, to bridge the gap between the EKF process language and BPEL. The difference between the supported patterns should give more insight into the conceptual differences at the M2 level of MDA (see Figure 5.2). We also consider the patterns not supported by the EKF process language, but which are supported by BPEL. For these patterns we can draw conclusions with respect to the limitations of the EKF process language when translating the EKF process models into BPEL code.

Patterns where traditionally the province of software design (e.g. widely referenced as object-oriented design patterns), suitable to provide more insight into the expressiveness of a language. Patterns have emerged from the PDA field as workflow patterns[1] and SOA field as interaction[2] patterns. The value of patterns lies in their independence of specific languages and implementation. A pattern, as conventionally specified, *'captures the essence of a problem, collects references by way of synonyms, provides real examples of a problem and even possible solution for implementation in terms of concrete technologies'*. In this thesis we follow a subset of the interaction and workflow patterns, for which the selection is based on the relevancy of the pattern with respect to the EKF process language. We adopted the classification of the interaction patterns and workflow patterns as presented in table 5.1 for our technical spaces transformation as included in Appendix C.

| **Classification of the pattern** |
|---|
| Interaction Patterns |
| Basic Control flow patterns |
| Advanced branching and synchronization patterns |
| Interaction Patterns |
| Multiple Instance Patterns |
| State Based Patterns |
| Cancellation Patterns |
| Termination Patterns |

Table 5.1: Classification of interaction [Barros et al., 2005b] & workflow [Aalst van der et al., 2003] patterns

The semantics of the EKF process are not defined explicitly by Everest, but are embedded in their process implementation. Therefore we need a way to define the semantics of the concepts in the EKF process models more formally. By making the semantics explicit, increases the understanding of the EKF process and promotes comparison of the language concepts. Efforts of [Ouyang et al., 2005, 2006, Aalst van der and Lassen, 2005, Mulyar, 2005] already provide some interesting insights in the semantics of BPEL. Nevertheless, we need to find a solution in BPEL, which complement the semantics of

[1]see also: http://www.workflowpatterns.com/

[2]see also: http://math.ut.ee/d̃umas/ServiceInteractionPatterns/

the patterns identified for the EKF process. More specific, we aim at finding a solution for which the semantics of the EKF process model match the solution expressed in terms of BPEL code. Implementing and simulating the BPEL solution of each patterns, should give a more clear understanding of the behavior and confidence of the correctness of the BPEL solution. Comparing concepts between languages is difficult, mainly when the formal semantics are not defined explicitly. Therefore we use Colored Petri Nets (CPN) [Aalst van der and Hofstede ter, 2002, Jensen, 1997] as a bases to formalize the semantics of the EKF process language, such that should give us a more clear understanding of the semantics of the concepts supported in the EKF process language. Especially, Petri-Nets provide a more formal bases to analyze the semantics of process concepts. Important is to notice that no previous attempts where undertaken by Everest to formalize their process syntax and semantics explicit. The results of the domain and technical spaces are therefore an important result for Everest, enabling a better basis to draw conclusions with respect to the expressiveness of the EKF process language.

## 5.2 Results of transforming the EKF process language into BPEL

Following the transformation approach of previous section we continue to present the results of the domain transformation (see Sections 5.2.1) and technical spaces transformation (see Section 5.2.2).

### 5.2.1 Results of the domain transformation

The aim of the domain transformation is to determine, which meta-model elements supported by the EKF process language, are directly supported through the syntax of BPEL. Our goal is to find the direct mappings between EKF process language and BPEL for all elements (e.g. sub-process, task and events, data, channel and role). Important is to find a BPEL solution for which the underlying implementation is threated similar compared to the EKF process implementation. The domain transformation should give us more insight into the syntactical differences between the EKF process language and BPEL, resulting in issues and challenges, when translating EKF process models into BPEL code.

Table 5.2 shows the results of the domain transformations, for which the following markings are considered: (+) a direct mapping of the meta-model element exist in BPEL; (+/−) a direct mapping of the meta-model element exist in BPEL, but this solution has limitation compared to the formal semantics of the EKF; (−) no direct mapping for

| **EKF Element** | **BPEL** |
|---|---|
| Process | - |
| Sub-process | - |
| Task | + |
| Start Timer Event | + |
| Start Message Event | + |
| Start Rule Event | +/- |
| Trigger Event | + |
| Intermediate Message Event | + |
| Intermediate Timer Event | + |
| Intermediate Rule Event | +/- |
| Cancel Event | + |
| Internal Event | + |
| End Event | + |
| Terminate Event | + |
| NOP Event | + |
| Cancel Sub-process | + |
| Notification Event | + |
| Role | + |
| Channel | + |
| Domain Instance | + |
| Flow | - |
| Decision-Point | - |

Table 5.2: Results of the domain transformation

the meta-model element exist in BPEL. Notice that this table gives an overview of the detailed study of the domain transformation as included in Appendix B. In preceding of this section we only discuss the issues and challenges, as a result of the domain transformation, which we need to overcome when translating the EKF process models into BPEL code (see Chapter 6).

For the task and event elements a direct mapping can be identified. Notice that for the (start and intermediate) rule events only a partial BPEL solution can be identified. BPEL only supports timer and message events and does not allow events driven by rules. Adopting the use of an external rule engine in the EKF, should provide a sufficient alternative to overcome this problem (see Section 7.1).

Transformation of the role, channel and data elements is more complex, as these capabilities are supported by the underlying WSDL and XSD definitions in BPEL. As a solution to we presented a domain transformation of the EKF domain model into XSD and EKF connectivity layer and WSDL (see Appendix B). This transformation is based on the ideas of [Carlson, 2001a,b,c] who has proposed an approach to bridge the gap between UML and XSD. In section 7.1 we use these observations to further address the results of the domain mapping, providing more insights into the best practices, when integrating an BPEL engine in the EKF.

Issues and challenges are identified when translating the process, sub-process, flow and decision-point elements into BPEL, as no direct mappings between these elements can be identified. For these elements the EKF process language and BPEL are considered to be two different classes of languages. BPEL is based on a block-oriented structure, while the EKF process language is based on a graph-oriented structure. The block-oriented property[3] of BPEL requires that the structured activities must have exactly one entry point and one exit point. This requirement results in issues, when translating the EKF (sub-)process, flow and decision-points into BPEL. The first problem occurs when translating the EKF (sub-)process with multiple entry and/or multiple exit point into the BPEL scope activity. The second problem is caused by the fact that the BPEL sequence, switch, pick, flow and while activities have a nesting structure. This nesting structure specifies that structured activities have a certain hierarchy, where activities are contained by other activities. This requirement is clearly violated by the EKF decision-points and flows, as the EKF process language allows modeling unstructured process concepts. The transformation of the technical spaces should further reduce the complexity of these issues, which is especially useful when translating the EKF process models into BPEL code.

For the issues and challenges described in this section we desribed above need to be resolved when translatating the EKF process models into BPEL code. These issues and challenges will therefore resolved in chapter 6 where we present an approach to transform the EKF process models into PBEL code.

### 5.2.2 Results of the technical spaces transformation

In the transformation of the technical spaces we aim at identifying patterns, which can be used to bridge concepts of the EKF process model and BPEL code. Our main objective is to provide an approach to transform EKF process models into BPEL code and therefore we need to evaluate if for all concepts supported by the EKF process language a BPEL solution exist. This implies that all patterns supported by the EKF process language must be covered. In extension of these patterns, we also evaluate patterns not directly supported by the EKF process language, but which are supported by BPEL. Including these patterns in this study fulfills our secondary objective, to draw conclusions with respect the limitations of the expressiveness of the EKF process language.

[3]The block-oriented property is the direct result of the XML based language concept of BPEL.

| **Pattern** | **EKF** | **BPEL** |
|---|---|---|
| Request/Reply (CP1) | + | + |
| One Way (CP2) | + | + |
| Synchronous Polling (CP3) | - | + |
| Message Passing (CP4) | + | + |
| Sequence (WP1) | + | + |
| Parallel Split (WP2) | + | + |
| Synchronization Merge (WP3) | - | + |
| Exclusive choice (WP4) | + | + |
| Simple Merge (WP5) | + | + |
| Multi-choice (WP6) | + | + |
| Synchronizing Merge (WP7) | - | + |
| Multi-merge (WP8) | - | +/- |
| Structured Discriminator (WP9) | +/- | - |
| Arbitrary cycles (WP10) | + | +/- |
| Implicit Termination (WP11) | + | + |
| MI without Synchronization (WP12) | + | + |
| MI with Priori Design-Time Knowledge (WP13) | - | + |
| MI with a Priori Run-Time Knowledge (WP14) | - | +/- |
| Structured loops (WP21) | + | + |
| Deferred Choice (WP16) | + | + |
| Interleaved Parallel Routing (WP17) | +/- | +/- |
| Milestone (WP18) | +/- | +/- |
| Cancel Task (WP19) | + | +/- |
| Cancel Case (WP20) | + | +/- |
| Cancel Region (WP25 (a & b)) | + | +/- |
| Explicit Termination (WP43) | + | + |

TABLE 5.3: Results of the technical spaces transformation

Table 5.3 shows the results of the technical spaces transformations, for which the following markings are considered: (+) a solution can be identified for the pattern in corresponding language, which match the formal semantics of the pattern; (+/−) a solution can be identified for the pattern in the corresponding language, but this solutions has limitations with respect to the formal semantics of this pattern[4]; (−) no solution can be identified for the pattern in the corresponding language. Notice that this table gives an overview of the results of the technical spaces transfromation as included in appencix C.

A conspicuous limitation of the EKF process language is the lack of support for synchronization of parallel branches, caused by a limitation of the EKF process engine. This limitation directly affect the lack of support for the patterns: CP3, WP3 and WP7. Nevertheless, the EKF process language support alternative approaches to synchronize

[4]As patterns are formalized in a language, we need to follow a predefined set of semantics of this language. This implies that minor differences between the formal CPN semantics and actual semantics of EKF process pattern are inevitable.

parallel branches (e.g. WP9, WP11, WP19 and WP43). However, this limitation does not result in considerable issues or challenges, when translating EKF process models into BPEL code. At this point BPEL is noticeable more expressive than the EKF process language. Still this observation results in limitations, as these concepts can not be modeled in the EKF process language. They will therefore never be available as a result after the transformation into BPEL. Therefore the EKF process language limits the result of the transformation, as BPEL does not support these patterns.

The pattern WP9 allows multiple parallel branches without specifying a single point of synchronization, this pattern is partially supported by the EKF process language. The completion of one parallel task could result in the continuation of the process, while the other task are synchronized at a point before its activation or after its completion. The EKF process language is considered to be a partial solutions as this pattern is only allowed at the level of the task in sub-process, not at the level of the (sub-)processes[5]. BPEL does not have any construct to support the semantics of this pattern, which is mainly caused by the fact that the link construct in combination with a join-condition are evaluated first, and not as required at the occurrence of the first positive link. This concept therefore results in considerable issues when translating EKF process models into BPEL code.

Although BPEL directly supports WP21, it has considerable limitation with respect to the support of WP10. This problem is mainly caused by the fact that BPEL does not allow unstructured loops as it is not possible to jump back to arbitrary parts in the process (i.e. only loops with one entry point and one exit points are allowed). As the lack of support of WP10 would cause considerable issues when translating the EKF process models into BPEL code. Therefore we follow an alternative implementation of WP10 as proposed by [Ouyang et al., 2006], they propose to take advantage of the event handler to accomplish the semantics of WP10. However, this solution is considered to be a partial solution to the problem, as it introduces some complexity issues when implementing this concept in terms of BPEL. These complexity issues directly arise from the technical limitations of the BPEL event handler. The event handler is exposed as a Web service as part of the process. BPEL does not allow that Web services that belong to a process are invoked from the process itself, therefore an external process is required to overcome this problem.

The EKF process language has considerable limitations with respect to the support of the multiple instance patterns. Although a solutions can be identified for WP12, the EKF process language does not support the patterns WP13 and WP14. This is mainly caused by the fact that the EKF process engine does not allow multiple instances of

[5]This is caused by a limitations of the EKF process language rather than is was intended this way.

tasks in the (sub-)process. BPEL directly supports WP12 and even a partial solution can be identified for WP13 and WP14. At this point BPEL is more expressive compared to the EKF process language. This limitation does not result in issues when translating the EKF process models into BPEL code. Nevertheless, Everest should diagnose this limitation as according to the workflow patterns multiple instances are important concepts for process modeling[6].

The transformation of the patterns WP17 and WP18 is more difficult as they allow more complex routings in the process. In the EKF process the rules control the state change in the process, such that conditions could be used to guard the activation of a task or sub-process. The order in which the task are performed are driven by the rule engine, where rules are used to drive the flow of the process. For both patterns we found alternative EKF process solutions, but they have limitations compared to the formal semantics. This is mainly caused by the fact that there are restriction when using these patterns in parallel threads. A similar solution for WP18 can be identified in BPEL, matching the semantics of the EKF process language. A BPEL solution matching the semantics of the EKF process can also be identified for WP17, but this solution requires the use of an external rule engine. As the solutions of BPEL complement the semantics of the EKF process language, we can conclude that these patterns will not cause issues for the transformation of EKF process models into BPEL code.

Limitation of WP19, WP20 and WP25 arise directly from the compatibility issues when translating the (sub-)process of the EKF process language into the BPEL scope activity, as already discussed in the domain transformation.

## 5.3 Issues & challenges as a result of the language transformations

In this section we summarize the issues and challenges which are a direct result of the domain and technical spaces transformation.

First, we consider the concepts which are supported by BPEL, but for which no EKF process solution exist. These limitations does not result in issues and challenges for the actual transformation, but directly affect the expressiveness of the transformation result. It is out of the scope of this thesis to resolve these limitation, but we provide Everest with insights into these limitations, such that its the responsibility of Everest

[6]Everest supports multiple instances only at a more fine-grained level, not part of EKF process modeling dimension.

to be aware of, and possible resolve[7], these limitations to take fully advantage of the expressiveness power of BPEL. Notice that the following limitations come directly from the domain and technical spaces transformation of previous section:

1. The EKF process language lacks for supporting synchronization of parallel branches, resulting in the lack of support of the patterns: CP3, WP3 and WP7.
2. The EKF process engine does not allow multiple instances of tasks in the (sub-) process, resulting the lack of support for the patterns: WP13 and WP14.

Second, we consider the concepts supported by the EKF process language, but for which no solution exist in BPEL. These concepts are interested as they result in considerable issues and challenges, which must be resolved in the preceding chapters where we: first, aim at translating the EKF process models into BPEL and second, integrating an BPEL engine in the EKF. The following issues have been identified when translating the EKF process language into BPEL:

1. BPEL requires the adoption of a rule engine to accomplish semantics of the rule based event types supported by the EKF process language.
2. The BPEL scope activity does not complement the semantics of the sub-process in the EKF process language, because it does not allow multiple entry and/or multiple exit points.
3. The structured activities in BPEL require a nesting structure, which is violated by the decision-point(s) in the EKF process language allowing unstructured process concepts. BPEL therefore requires an alternative implementation of WP10, introducing some complexity issues.
4. BPEL does not support WP9. This pattern can be no means be translated into BPEL as no solution or partial solution exist in BPEL.
5. The EKF process allows multiple entry and exit points for an (sub-)process, which result in problems when translating the (sub-)process supported by the EKF process language into the BPEL scope activity. This result in considerable issues when translating the patterns: WP19, WP20 and WP25 into BPEL.
6. Closing the gap between the role, channel and data in EKF and BPEL requires the adoption of WSDL and XSD in the EKF.

[7]Resolving theses issues requires language extensions in the EKF process language and engine implementation.

In this chapter we provided an overview of the results of the domain and technical spaces analysis when closing the gap between the EKF process language and BPEL. We only focus on aspect of the differences between the language context and concepts from which we derived the issues and challenges that must be considered when transforming the EKF process language into BPEL. In the following chapter we need to resolve these issues and challenges to provide an approach to transform the EKF process language into BPEL code. Which is different from this chapters as we present a plan to transform EKF process models (composed with the EKF process language) into BPEL code (composed of BPEL).

Chapter 6

# Transform EKF process models into BPEL code

In this chapter we continue answering the research question: *'How to make the Everest Knowledge Framework more interoperable by adoption of BPEL?'*, but in preceding of the previous chapter we focus on answering the partial question: *'How EKF process models can be transformed into BPEL code?'*. First, we present an approach to transform the EKF process models into BPEL code (see Section 6.1). Second, we discuss the issues and challenges with respect to the completeness and correctness properties of our proposed solution (see Section 6.2).

## 6.1 Approach to transform EKF process models into BPEL code

In this section we describe the approach, composed of four steps (see Figure 6.1), used to transform the EKF process models into BPEL code.

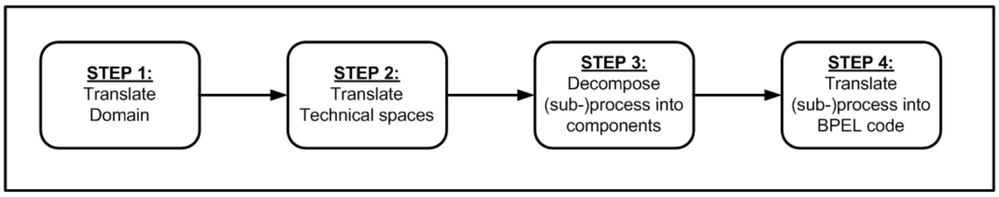

STEP 1: Translate Domain
STEP 2: Translate Technical spaces
STEP 3: Decompose (sub-)process into components
STEP 4: Translate (sub-)process into BPEL code

FIGURE 6.1: Steps of the transformation approach.

In the first step *translate domain* we use the results from the domain transformation (see Appendix B) to decompose the EKF (sub-)processes into components ($C$), for which each component can directly be translated into the corresponding BPEL code.

The decision-points and flows are not directly supported through the domain transformation. Therefore we apply the transformations of the technical spaces to reduce the complexities introduced by the BPEL limitations. In the second step *translate technical spaces* we decompose the (sub-)process into components ($C$), by identifying and translating each concept from the EKF process model into the corresponding BPEL code.

In the third step *translate unstructured (sub-)process* we need to resolve the issues and challenges which are not covered by the previous step. We address the limitation of BPEL in case patterns are not directly supported by BPEL and the patterns for which no solution exist in BPEL[1]. Notice that for the limitations for the cancellation patterns (WP19, WP20 and WP25) are resolved in the following step, as the origin of the problem is caused by the limitations of the BPEL scope activity.

In the fourth step *translate (sub-)process into BPEL* we need to resolve the issues and challenges when translating the EKF (sub-)process into the BPEL scope activity. We also need to address to transformation of additional events which belong to the EKF (sub-)process, namely: trigger events, exception events and cancel events.

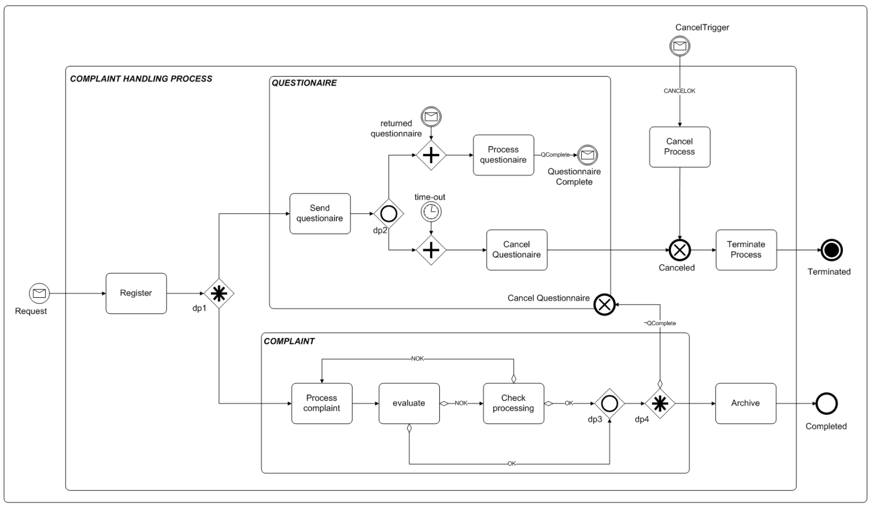


FIGURE 6.2: Running example: a complaint handling process.

To give more insight in how each translation step is performed, we present a running example[2] as shown in figure 6.2. This example describes a complaint handling process, where first a complaint is registered (task registered), then in parallel a questionnaire is sent to the complaint (sub-process questionnaire) and the complaint is processed (sub-process complaint). If the complainant returns the questionnaire within two weeks (event returned questionnaire), the task process questionnaire is performed. Otherwise,

[1]These are the patterns WP9 and WP10

[2]This example is the example from [Ouyang et al., 2006], but modeled in terms of the EKF process language

the result of the questionnaire is discarded (event time-out). In parallel the complaint is evaluated (task evaluate). Based on the evaluation result, the processing is either completed or continues the task check processing. If the check result is NOK, the complaint requires re-processing, otherwise the sub-process questionnaire is canceled (only if it was still active) and the task archive is performed. Finally, the entire process can be canceled through a cancel trigger, but only if the condition CANCELOK has been satisfied.

### 6.1.1 STEP 1: Translate the domain

In the first step we identify the elements in the EKF process, in a way that each element can directly be translated, following the domain transformation as included in Appendix B. For the transformation of the elements we introduce a the function $\Psi_i(C_m, C_n)$, for which a certain component $(C_m)$ from the EKF process model can be translated into the BPEL code $(C_n)$, **iff** the component is an element of type $(i)$. For each element for which this function can be applied the BPEL code is folded into a component $C_x$. Notice that we consider only the elements which are directly supported by BPEL: the (sub-)processes, flows and decision-points are not considered at this stage. *Folding* a component in this step implies that the element references to the translated BPEL code by means of the component $C_x$. This step is repeated until no more elements in the EKF process model can be identified for which a direct mapping exist.

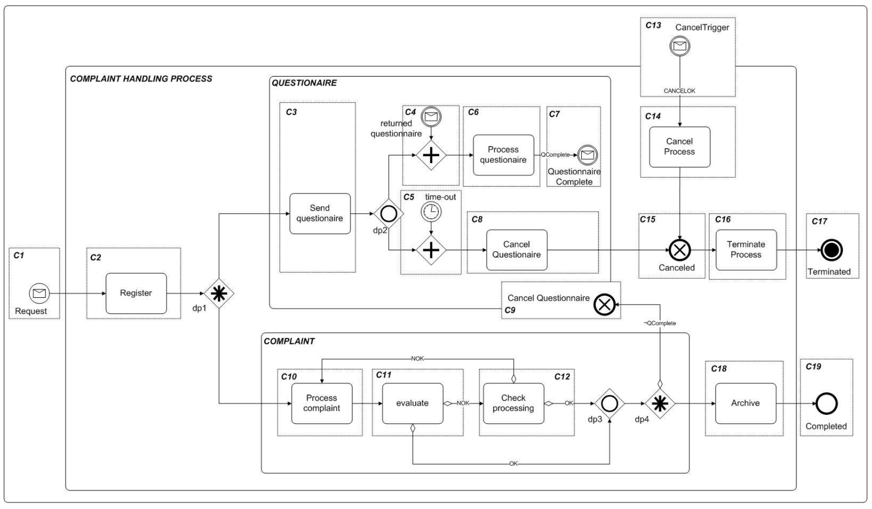


FIGURE 6.3: Transformation of complaint handling process (Cont. STEP1).

By applying the first step to the complaint handling example, we can identify the components $C_1, C_2, ..., C_{19}$ (see Figure 6.3). Notice that each component references to a BPEL code fragment, which are translated using the domain transformation of Appendix B.

Take for example component $C_3$ is and element of the type task, which can (according to appendix B) be translated into a BPEL invoke activity as presented in listing 6.1.

```
<invoke operation="Send Questionaire"/>
```

Listing 6.1: BPEL code of C3 in complaint handling example

### 6.1.2 STEP 2: Translate the technical spaces

In the second step we identify the components in the (sub-)process, in such a way that each component is translated into a concept directly supported by means of the technical spaces transformation as included in appendix C. For the transformation of the patterns we introduce the function $\delta_i(C_m, C_n)$, for which a certain component ($C_m$) from the EKF process model can be translated into the BPEL code ($C_n$), **iff** the component is a pattern type ($i$). This step is repeated for each (sub-)process until no more patterns can be identified. Notice that we only consider the patterns, which are directly supported by BPEL. The patterns WP9, WP10, WP19, WP20, WP25 are therefore not considered at this stage.

```
<scope="Cmax">
 <!--  Translation of the maximal component Cmax -->
</scope>
```

Listing 6.2: Event Action Rules in BPEL

To reduce the complexity of the transformation problems we aim at finding a maximal component ($C_{max}$), which groups several components into a new component supporting a certain concept. A maximal component is only allowed, when the grouping of components does not introduce an arbitrary cycle or crosses the boundaries of a (sub-)process. By translating a maximal component into BPEL, we increase the readability of the resulting BPEL code. More importantly we reduce the complexity of the problem, before we try to resolve the issues. Finally, the BPEL code for the maximal component is folded into a BPEL scope activity (see Listing 6.2). *Folding* a component in this step implies that a BPEL scope activity substitutes the translated BPEL code for the maximal component, such that we can reduce the complexity of the problem by replacing the maximal component $C_x$ by a collapsed sub-process $C_y$. This step is recursively repeated until no more maximal components can be identified within the (sub-)process.

Applying the second step to the complain handling example, we can identify the components $C_{20}, C_{19}, ..., C_{30}$, which directly correspond to the patterns from the technical spaces transformation (see Figure 6.4). Several maximal components can be introduced $C_{31}, C_{32}..., C_{34}$. Take for example $C_{21}$ and $C_3$ are grouped into a new component $C_{32}$ (see

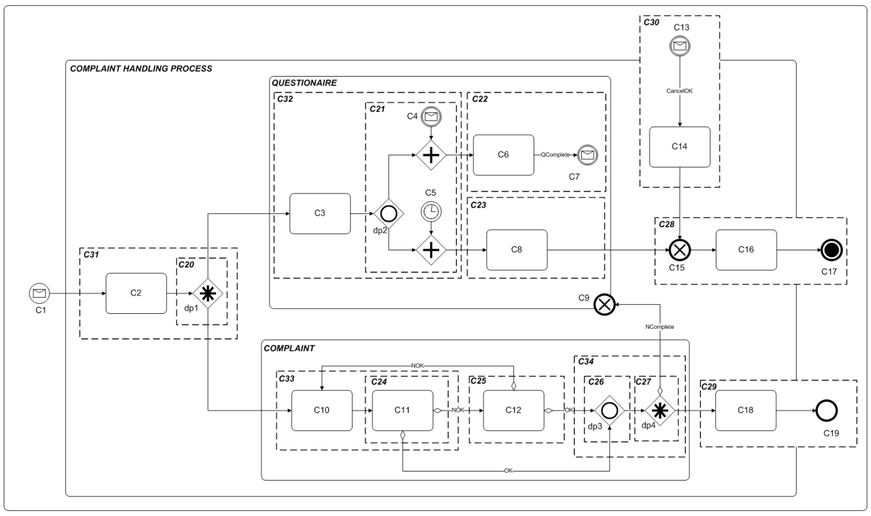


FIGURE 6.4: Transformation of complaint handling process (Cont. STEP2).

Listing 6.3) as they are both part of a sequence pattern and do not cross the boundaries of the sub-process questionnaire or introduces an arbitrary cycle.

```
<scope name="C32">
 <sequence>
  <invoke operation="Send Questionaire"/>
  <pick>
   <onMessage operation="Returned...">
    <invoke operation="Start(C22)"/>
   </onMessage>
   <onAlarm name="time-out">
    <invoke operation="Start(C23)"/>
   </onAlarm>
 <sequence>
</scope>
```

LISTING 6.3: BPEL code of C21 in the complaint handling example

### 6.1.3 STEP 3: Decompose the (sub-)process into components

This step is only required when the resulting (sub-)process after the second step still contains issues which must be resolved (e.g. contains arbitrary cycles or unstructured process components). To resolve these issues we follow the approach of [Ouyang et al., 2006]. In this approach each maximal component as identified in the previous step must be translated into an event action ($E\{A\}$) rule. These $E\{A\}$ rules can be translated into the *onMessage* event handler in BPEL, where the event ($E$) corresponds to the start point of the maximal component and the action ($A$) specifies the folded BPEL code as derived in the previous step. Listing 6.4 gives an overview of the BPEL code used for the translation of the event action rule ($E\{A\}$). The ($E\{A\}$) rule is constructed for each maximal component contained by the (sub-)process.

```
<onMessage="Start(x)">
 <!--  Translation of the code from step 2 -->
</onMessage>
```

LISTING 6.4: Event Action Rules in BPEL

Following the third step we can translate the sub-process complaint into BPEL code (see Listing 6.5). This sub-process is composed of the maximal components $C_{25}, C_{26}$ and $C_{29}$, which can not be further decomposed. Each of the resulting components are translated into event action rules.

```
<!--------------- Event Action translation of C33 --------------------->
<onMessage operation="Start(C33)">
 <scope name="C33">
  <sequence>
   <invoke operation="Process Complaint"/>
   <invoke operation="Evaluate"/>
   <switch>
    <case condition="NOK">
     <invoke operation="Start(C33)"/>
    </case>
    <otherwise>
     <invoke operation="Start(C25)"/>
    </otherwise>
   </switch>
  </sequence>
 </scope>
</onMessage>
<!--------------- Event Action translation of C25 --------------------->
<onMessage operation="Start(C25)">
 <scope name="C25">
  <sequence>
   <invoke operation="Check Processing"/>
   <switch>
    <case condition="NOK">
     <invoke operation="Start(C33)"/>
    </case>
    <otherwise>
     <invoke operation="Start(C34)"/>
    </otherwise>
   </switch>
  </sequence>
 </scope
</onMessage>
<!--------------- Event Action translation of C34 --------------------->
<onMessage operation="Start(C34)">
 <scope name = "C34">
  <flow supressJoinFailure="yes">
   <invoke operation="Start(C7) joinCondition="NCOMPLETED"/>
   <invoke operation="End(C34)"/>
  </flow>
 </scope>
</onMessage>
```

Listing 6.5: Translate the sub-process complaint into BPEL

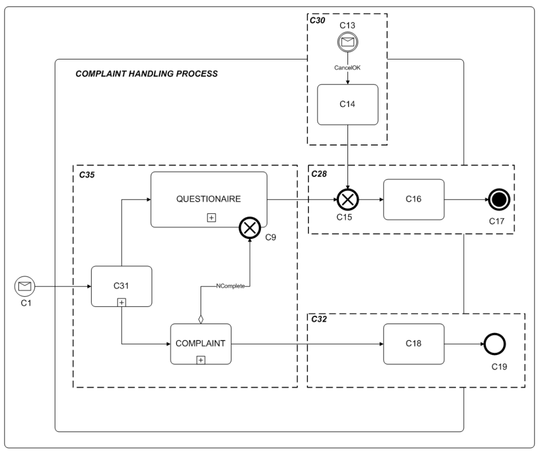


Figure 6.5: Transformation of complaint handling process (Cont. STEP3).

### 6.1.4 STEP 4: Translate the (sub-)process into BPEL code

In this step we need to resolve the limitations when translating a (sub-)process into a BPEL scope activity. To accomplish this we introduce a scope activity resembling the (sub-)process containing the result (event action rules) of the previous step as presented in listing 6.6. The initial component $C_x$ in the sub-process is triggered by the $Start(x)$ (see Line 8 in Listing 6.6 ) and the scope waits until it receives and $End(y)$ (see Line 9 in Listing 6.6) marking the completion of the scope activity. This step must be performed for all (sub-)processes, which contain an arbitrary cycle or unstructured components. By following this approach we can overcome the issues of multiple entry and exist points for the (sub-)process and containment of unstructured process concepts.

There still remains a considerable problem when the EKF process model contains the pattern WP9. There is no BPEL translation available to support this pattern, such that we can by no means resolve this problem through BPEL. Resolving this problem therefore requires either re-modeling (remove all WP9 pattern before the transformation) or transform the entire pattern into a scope activity contained by a question mark (?). The last solution requires intervention of a BPEL specialist, to resolve the question marks before the BPEL code can be deployed. In the case study (see Appendix C) provides an example of the transformation of a EKF process model including a WP9 pattern.

The (sub-)processes containing trigger, exception or cancel events (e.g. WP19, WP20, WP25), require additional event and/or fault handlers in the BPEL scope activity.

Therefore we need to translate the event types, which specifically belong to EKF (sub-)process, into the corresponding part of the event and fault handler of the BPEL scope activity. The fault handers are required when the instance of a (sub-)process are explicitly terminated (e.g. WP19, WP20 and WP25a) and continue processing from a single point forward. The fault handlers are required for trigger events, cancel events and cancel sub-process event, while exception event only result in additional event handlers.

```
<scope name="(sub-)process">
 <faultHandler>
  <!-- transformation of the fault handlers -->
 </faultHandler>
 <eventHandler>
  <!-- translation of the event handlers -->
 </eventHandler>
 <invoke operaton="Start(x)"/>
 <receive operation="End(y)"/>
</scope>
```

Listing 6.6: Arbitrary Cycle in BPEL

The component $C_{31}$ is the initial component in the process complaint handling, such that this is the first component that must be activated in the sub-process complaint (see Line 69 in Listing 6.7). The component $C_{29}$ is the final component in the process complaint handling, such that this component is responsible to enable the completion of the scope complaint (see Line 69 in Listing 70).

For the components $C_{15}$ and $C_{11}$ we introduce respectively two fault handlers: fCANCEL, which terminates all process instances in case the process was explicitly canceled; and fCANCELTRIGGER, which is required to terminate all process instances in case the Cancel Trigger event has occurred and the condition CANCELOK was satisfied. Notice that the process complaint also includes a typical WP19 pattern, allowing the termination of the sub-questionnaire can be terminated from within the sub-process complaint. To accomplish this, one must include an additional fault handler in the scope activity resembling the sub-process questionnaire.

```
1  <scope name="COMPLAING HANDLING PROCESS">
2   <faultHandler>
3    <!--------------- Fault Handler for the cancel event ----------------->
4    <catch faultName="fCANCEL">
5     <invoke operation="Start(C29)"/>
6    </catch>
7    <!--------------- Fault Handler for the cancel event ----------------->
8    <catch faultName="fCANCELTRIGGER">
9     <sequence>
10     <invoke operation="Cancel Process"/>
11     <throw faultName="fCANCEL"/>
12    </sequence>
13   </catch>
```

```
</faultHandler>
<eventHandler>
<!--------------- Event Action translation of C30 --------------------->
<onMessage operation="CANCELTRIGGER">
 <scope name="C30">
  <switch>
   <case condition="CANCELOK">
    <throw faultName="fCANCELTRIGGER"/>
   </case>
   <otherwise>
    <empty/>
   </otherwise>
  </switch>
 </scope>
</onMessage>
<!--------------- Event Action translation of C31 --------------------->
<onMessage operation="Start(C31)">
 <scope name="C31">
  <sequence>
   <receive operation="Request" createInstance="yes"/>
   <invoke operation="Register"/>
   <flow>
    <invoke operation="Start(QUESTIONAIRE)"/>
    <invoke operation="Start(COMPLAINT)"/>
   </flow>
  </sequence>
 </scope>
</onMessage>
<!--------------- Event Action translation of C28 --------------------->
<onMessage operation="Start(C28)">
 <scope name="C28">
  <sequence>
   <invoke operation="Terminate Process"/>
   <terminate name="Terminated"/>
  </sequence>
 </scope>
</onMessage>
<!--------------- Event Action translation of C29 --------------------->
<onMessage operation="Start(C29)">
 <scope name="C29">
  <sequence>
   <invoke operation="Archive"/>
   <invoke operation="End(COMPLETED)"/>
  </sequence>
 </scope>
</onMessage>
<!--------------- Event Action translation of QUESTIONAIRE------------->
<onMessage operation="Start(QUESTIONAIRE)">
 <!-- code for Questionaire of previous step-->
</onMessage>
<!--------------- Event Action translation of COMPLAINT---------------->
<onMessage operation="Start(COMPLAINT)">
  <!-- code for Complaint of previous step -->
</onMessage>
</eventHandler>
```

```
 <invoke operaton="Start(C31)"/>
 <receive operation="End(COMPLETED)"/>
</scope>
```

Listing 6.7: Translation of the running example into BPEL

## 6.2 Correctness and completness properties

In this section we primary discuss the correctness and the completeness properties of the transformation result, when following our transformation approach. Based on a mortgage process from the Everest practice, we present a case study for which we derive the BPEL code following our transformation approach (see Appendix D). Because the first and second step of our transformation approach is straightforward and is already explained through the running example, we only describe the results of the third and fourth step for this case study. We have identified that we can transform the complete mortgage process into BPEL, following the proposed transformation approach. Still we must consider some limitations with respect to our proposed transformation approach, as this approach depends on the results from the domain and technical spaces transformation of the previous chapter.

### 6.2.1 Can we translate all EKF process models into BPEL?

The evaluation of the completeness should answer the question: *'Can we translate all EKF process models into BPEL?'*. Answering this question is particular interesting when using the transformation approach as a general approach to transform EKF process models into BPEL code.

Validation of the completeness of the technical spaces is impossible as no absolute notion or insight is available to us to draw conclusions about the completeness. This is mainly caused by the fact that our transformation approach is based on the assumption that the technical spaces are concidered to be complete. As this assumption is clearly violated in case concepts are 'overlooked'[3], which require extensions in the domain and/or technical spaces transformation of appendix B and C. Still we can say with considerable certainty that we have reached a certain completeness in our approach, as the case study is considered to be a representative EKF process model from the Everest practice. Therefore we assume that this process models covers all elements and concepts supported by the EKF process language.

[3]Models consist of concpets which are not covered by our technical spaces

### 6.2.2 What can we say about the correctness of the transformation?

For the evaluation of the correctness we aim at answering the following two questions: *'Is the'result of the transformation correct BPEL code?'* and *'Is the resulting BPEL code semantical equivalent to semantics EKF process model from which the transformation was made?'*. Notice that we are not interested in the correctness of the BPEL language itself, but only the limitations of the target BPEL code as a result of our transformation approach.

The validation of the correctness of our transformation approach requires the evaluation of the correctness of the initial EKF process models and resulting BPEL code. The EKF process language and BPEL are not based on formal languages like: Petri-Nets, $\pi$-calculus or process algebra. Validation of the correctness based on a formal approach is difficult and out of the scope of this thesis. Therefore we need another approach to draw conclusions about the correctness of the transformation result. efore we need another approach to draw conclusions about the correctness of the transformation result.

The EKF process model from our case study is an operational solution, which has passed various testing stages before its delivery. This should give use some confidence with respect to the correctness of this EKF process model. The EKF process models from which the transformation is derived, must be correct otherwise the resulting BPEL code will certainly be incorrect.

For the validation we used Oracle BPEL designer and Process Manager to check if each concepts (from the technical spaces) can actually be implemented in BPEL and determine based on simulation if the result produces the desired behavior. Following this approach we can say that we have reached certain confidence with respect to the semantical correctness of the individual patterns as included in appendix C. As the EKF process is composed of these patterns, we assume that the transformation result is considered to be correct under the condition that all individual patterns are considered to be correct. Providing an actual implementation of case study in BPEL should increase the certainty of the correctness. However, this is left to further research (see Section 7.4) as it was not possible to accomplish the implementation of BPEL and gather the required data, within the project time constraint. Especially the requirement of integrating the BPEL engine and EKF rule engine makes it difficult to accomplish this in the limited amount of time.

At this point, we provided an approach to close the gap between the EKF process models and BPEL code, toward a solution to the problem as identified earlier in this report (see Chapter 1). In the following chapter we start a discussion as a reflection and positioning of our work.

# Chapter 7

# Discussion

Everest is interested in the issues and challenges which can deteriorate the competitive strength of the development environment like the EKF. Specifically for Everest it is important to keep-up with the latest developments, toward and extensible and interoperable EKF. In the previous chapters we provided a solution to the problem as identified in chapter 1. As for every solution we need to consider limitations and trade-off that have to be made when adopting a specific solution. In this chapter we therefore reflect and position our work by starting a discussion of the best practices which must be considered when adopting our proposed solution. This discussion should reveal insights, which are not addressed by answering the research questions as done in the previous chapters, but which are important in the context of Everest. In this chapter we therefore focus on the reflection and positioning of our work, by answering the question: *'What are the best practices which must be considered when adopting our proposed solution?'*.

To answer this question we first discuss the best practices which must be considered when integrating a BPEL engine into the EKF (see Section 7.1). Integration of a BPEL engine in the EKF is important as our solutions has identified that some aspect of the EKF are are not covered by BPEL, such that integration of a BPEL engine and the EKF is required to resolve these issues. Second, we evaluate the EKF process language compared to alternative process modeling languages (see Section 7.2). This should allow us to derive conclusions with respect to the competitive strength of the EKF process language. Third, we position our work providing an overview of the related work (see Section 7.3). Finally, we provide the objectives for further research that directly arise from our work (see Section 7.4).

## 7.1 What best practices must be considered when integrating a BPEL engine in the EKF?

In order to take fully advantage of BPEL, Everest should consider the integration of a BPEL engine in the EKF. This implies that a BPEL engine should be integrated in the enterprise architecture of the EKF. The integration of a process engine in the enterprise architecture can best be illustrated through the WfMC Reference Model (see Figure 7.1). This reference model specifies five interfaces, which must be considered when integrating a process engine in the enterprise architecture.

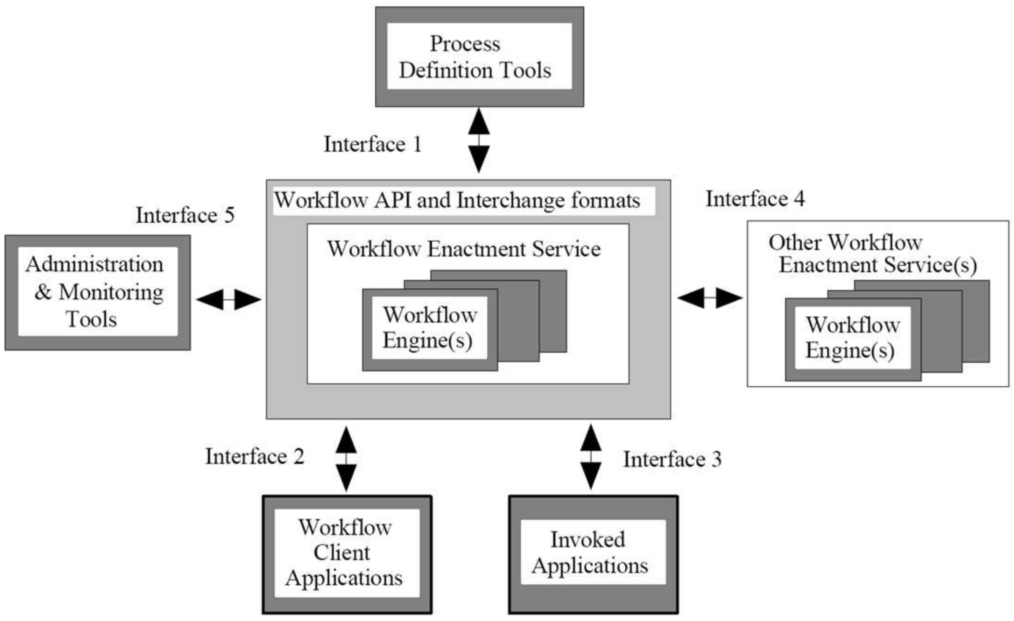

Process Definition Tools
Interface 1
Workflow API and Interchange formats
Workflow Enactment Service
Workflow Engine(s)
Interface 5
Administration & Monitoring Tools
Interface 4
Other Workflow Enactment Service(s)
Workflow Engine(s)
Interface 2
Interface 3
Workflow Client Applications
Invoked Applications

FIGURE 7.1: WfMC Reference Model [Hollingworth, 1995].

### 7.1.1 Interface 1

Integration of a BPEL engine in the EKF requires that the EKF process engine capabilities are replaced by the BPEL engine capabilities. This implies that the process definition tools for BPEL can be used by the business engineer or software engineer to compose BPEL process definitions. Interface 1 therefore conciders an interchange format and Application Programming Interface (API), which can support the exchange of the process definition across a variety of tools and engines. This interface is directly supported when adopting BPEL as the BPEL process definition can be used as a language to interchange the process definition. An increasingly number of tools and engines are available supporting BPEL, which is a direct result of the popularity of BPEL. For

Everest the implementation tools could also be of interest as our proposed solution still requires manual implementation of BPEL.

### 7.1.2 Interface 2

The external activities in BPEL are exposed as Web service, so that the obvious business activity of a human interaction (interface 2) is not covered by BPEL standard. The implementation of BPEL is primarily designed to support automated business processes based on Web services. However, the spectrum of activities that make up general purpose business processes is broader than this, because people often participate in the execution of business processes. The worklist manger is therefore not supported by the BPEL standard, but this is mostly covered by its implementation. Standards like BPEL4People propose extensions to BPEL in terms of scenarios involving people within the BPEL process [IBM and SAP, 2005]. BPEL does not specify guidelines for the engine and worklist implementation. The Workflow Management Application Programming Interface (WAPI) [WFMC, 1998] could fill this gap, as it provides a standard for specifying the required interfaces when implementing a workflow engine or worklist manager.

### 7.1.3 Interface 3

The challenge of integrating a BPEL engine in the EKF lies in respectively the interfaces 3 of the WfMC reference model, which specifies the interaction with the logical components in order to complete a task in the process. In chapter 3 we have identified that both interfaces are provided by the EKF by means of ESB channels by means of the connectivity layer. The integration of a BPEL engine in the EKF for the interfaces 3 can therefore be seen as a continuum from very tight, to very loose (see Figure 7.2).

*Tight integration* specifies that the BPEL engine and EKF are integrated such that they are exposed as a single point of processing. The main drawback of this approach is that applications must interact with a combined BPEL engine and EKF, also if it only needs to interact with one of them. This approach lacks for scalability, as this approach does not allow that the EKF and process engine are independently scalable. In a *loosely coupled integration* approach the BPEL engine and the EKF operate completely independent and are integrated on the level of the client. This integration approach lacks for flexibility toward change, as changes require re-deployment of the client application. A *loosely coupled integration* approach lies in-between a tight and loose integration, taking advantage of the enterprise architecture as enabling a service based integration approach. Adoption of a more loosely coupled approach should increase the scalability

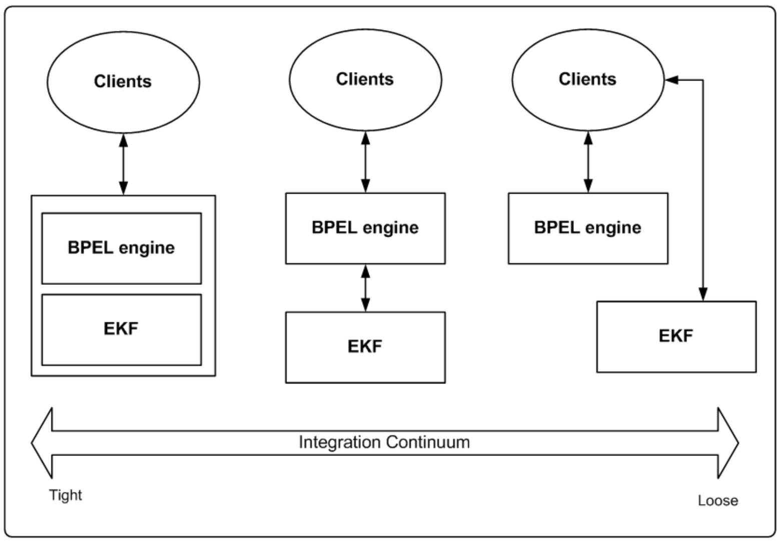
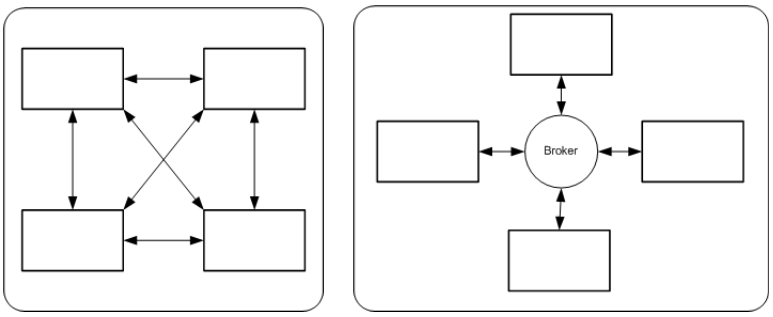


FIGURE 7.2: Continuum of EKF and BPEL engine integration from [WfMC et al., 2003].

and separation of concerns between the BPEL engine and EKF engine components, promoting standardized interfaces between the logical components. This is an advantage, as Web service technologies are the leading standard for specifying these interfaces in the enterprise architecture. A drawback of this approach could be performance issues, due to the fact that Web service protocols are known to have performance issues compared to tight integration. Nevertheless, we think that a loosely coupling approach is most suitable for the engine-to-engine integration of BPEL and the EKF.



(a) Point-to-point integration (6 connections)

(b) Broker integration (4 connections)

FIGURE 7.3: Point-to-point versus broker integration from [Harvey, 2005].

A loosely coupled integration of a BPEL engine and the EKF requires adoption of Web service technology in EKF. Web service technology is the backbone of BPEL and provides interoperability across various applications and platforms by facilitate the publishing of applications as Web services. Following the loosely coupled integration strategy we can continue to discuss the best practices which must be considered when closing the gap between a BPEL engine and enterprise architecture of the EKF.

First we address the complexity issue of the integration, maintaining the connections on a point-to-point basis (see Figure 7.3(a)). The drawback of point-to-point connections between the BPEL engine and the EKF results in complexity issues, because of the increasing number of interfaces required for the point-to-point solutions. Take for example: $N$ components, the number of point-to-point connections required is $N * (N-1)/2$. Therefore we propose the use of a message broker (see Figure 7.3(b)). The message broker reduces the number of conncetions and decreasing the complexity of number of interfaces which need to be considered when integrating $N$ components.

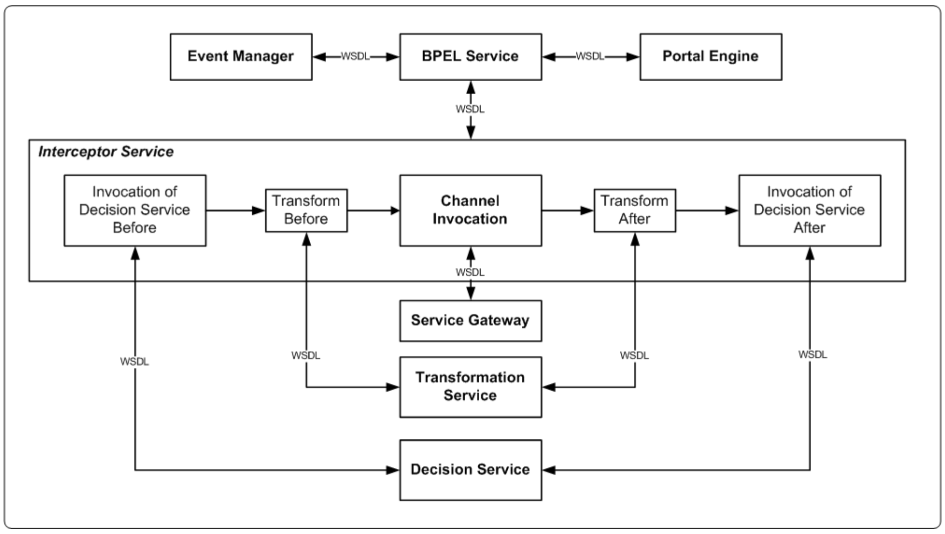


FIGURE 7.4: Integration of EKF and BPEL engine based on idea from [Rosenberg and Dustdar, 2005a].

To achieve the message broker following a loosely coupled integration approach, it requires the adoption of Web services in the EKF connectivity layer. In this way the generalized channel in the EKF should be exposed as Web services as proposed by [Debevoise, 2005]. This allows that all the specialized channel types are also exposed as Web services: BPEL service, interceptor service, decision service and transformation service (see Figure 7.4).

The interceptor service acts as a broker for the integration of the invoked applications on interface 3. Directly invoke the rules from BPEL results in complexity issues (increasing the number of activities to $2 * N$) as described by [WfMC et al., 2003]. A solution to this problem is presented by Rosenberg and Dustdar [2005a], where they assign this task to the interceptor service. The role of the interceptor service is to intercept each incoming and outgoing BPEL service call to automatically apply the rules. The interceptor service applies through a decision service, before and after the invocation of the channel task through the service gateway. Linking the BPEL activities to the rules can be accomplished through a mapping file[1].

Redeployment of the process is required, when BPEL is responsible for the binding to the end points and the underlying service change. Maintenance issues arise when the number of participating services in the process increase, which requires maintaining the binding of an increasingly number of services. To overcome this problem, Kuster and Konig-Ries [2007] proposes to dynamically bind the service end point at run-time through the service gateway rather than explicitly define these end points in the BPEL process. The service gateway should expose a generic function of the EKF by means of Web services (e.g. interaction, transformation, decision and data, etc.). The service gateway implies that channels in the EKF connectivity layer must be exposed as WSDL operations. Each operation in the WSDL, need XSD's to describe the layout of the XML document for its input and output variables (transformation of the EKF domain model into XSD and channels into WSDL is described into more detail in appendix B).

As each task is exposed as a Web service and BPEL specifies the orchestration of these Web services. Therefore it is required that the output variables of one Web service must be translated into the input variables of the preceded invoked Web service. As these transformations must be performed before and after the actual invocation of the task they are considered to be part of the interceptor service. The transformation service is introduced to transform one XML data format into another XML data format, by providing an XSLT transformation definition. However, this solution has a drawback as it is time consuming to create and complex to maintain the increasing number of transformation definitions. Solution to this problem can possibly be found in the Java Meta-modeling Protocol (JUMP)[2], which offers a framework and structured meta-language to consolidate multiple models into a single view [Jumper, 2006]. This view captures the required integration details for correctly cleansing and mapping the domain attributes and expose them as Web services.

The introduction of the decision service exposes the distinctive features of the rule engine as a Web service. The concept of such a service is to derive knowledge from a set of rules and data on its input [Martin, 2006]. Following the approach of Rosenberg and Dustdar [2005b], we can define a decision service as a triple $< R, I, O >$, where $R$ is the ruleset composed of rules and $I$ and $O$ are respectively the input and output variables of the ruleset. Considering a ruleset as an input parameter of a decision service requires the use of a rule language defining the behavior of the actual Web service (referred to as semantic Web service) [Gasevic et al., 2006]. The development of rule language

[1]This mapping file simply links the invoked activity to the rules which need to be applied, promoting reuse of rules

[2]see: http://en.wikipedia.org/wiki/JUMP_-_Java_metamodel_protocol

is work in progress (e.g. Rule Meta Language (RuleML)[3], Rule Interchange Format (RIF)[4] and Production Rule Representation (PRR) [OMG, 2006]), but no defacto rule language has been adopted by the industry at the time of writing. Therefore we consider the approach of Rosenberg and Dustdar [2005a], which proposes to expose each ruleset as an independent Web services. In this way the Web service does not require the ruleset as an input variable of the Web service, but exposes the entire ruleset as an independent WSDL operation. Maintaining the increasing number (versions) of Web service, when changes to the rules occur is the main concern of this approach. To overcome this problem one could follow the approach of Orriens et al. [2003], Schmidt [2002], proposing to generate WSDL operations directly from the ruleset definitions.

As each BPEL process is itself exposed as a Web service, it is possible for the portal engine and event manger in the EKF to directly interact with the process through the invocation of the process Web services.

### 7.1.4 Interface 4

This interface defines the way how multiple enactment services can be integrated. This interface is directly supported by BPEL, by exposing each (sub-)process and events as a Web service. Various implementations of BPEL can interact by means of invocation of services across the deployed BPEL processes.

### 7.1.5 Interface 5

This interface includes standards for administration and monitoring function which allow one vendor management application to work with other engine(s). This promotes a common interface which enables several workflow services to share a range of common administration and monitoring functions. Take for example an interface for the interchange of overall status and metric information. Notice that it is outside the scope of BPEL standard to define such an interface, but various BPEL implementations already cover some implementation in terms of an API. This implies that the way this information is stored in terms of the domain model could be expressed in terms of standardized metric vocubularies (e.g. Business Motivation Model (BMM) or Business Process Definition Model (BPDM)). The interface must be exposed as a specific service and protocol which is used to retrieve the status and statistical information at run-time.

[3]see: http://www.ruleml.org/

[4]see: http://www.w3.org/2005/rules/wg/charter.html and also [W3C, 2005]

## 7.2 EKF process language versus formalized process modeling languages

The EKF process language is specifically developed for modeling the processes in the EKF and does not support a formal basis in process modeling. This clearly deteriorates the competitive strength of the EKF process language as automated validation and generation of BPEL from EKF process models is difficult and time consuming. The adoption of a more generic and formal modeling language seems most suitable to overcome this problem.

Various modeling language exist which have a formal background in either Petri-nets, $\pi$-calculus or process algebra: Activity Diagrams (AD) [OMG, 2005], Business Process Modeling Notation (BPMN) [White, 2004, Russell et al., 2006] and Yet Another Workflow Language (YAWL) [Aalst van der and Hofstede ter, 2003]. We compare the expressiveness by performing an evaluation of the supported workflow patterns. Various efforts of Decker and Puhlmann [2007], Russell et al. [2006], Wohed et al. [2005a, 2006, 2005b] have already addressed the evaluation worklow patterns for: BPMN, AD and YAWL. An overview of their results is presented in table 7.1[5].

Table 7.1 makes it clear that the EKF process modeling language has several limitations supporting some of the workflow patterns. The main limitations of the EKF process language are the support for: WP3, WP7, WP8, WP13 and WP14, which are noticeable directly supported by AD, BPMN and YAWL. In chapter 6 we already have identified that taking advantage of the expressiveness of BPEL requires extensions in the EKF process language. Alternatively Everest could consider to adopt a more formal modeling language, to increase the expressiveness and adopt a more formal approach for modeling. Adopting AD, BPMN or YAWL could therefore increase the competitive strength of Everest, promoting automated validation and transformation of models. It is outside the scope of this thesis to reccomend specific modeling languages, but it is considered the responsibility of Everest to be aware of the limitations of the EKF process modeling language. In the following section we discuss the related work, including efforts of transforming more generic modeling languages into BPEL.

## 7.3 Related work

Since BPEL is increasingly supported by various vendor implementation, it has become interesting to link a modeling language with BPEL. Notice that BPEL more closely

[5]+directly supported; +/- partially supported; - not support

| **Pattern** | **EKF** | **AD** | **BPMN** | **YAWL** |
|---|---|---|---|---|
| Sequence (WP1) | + | + | + | + |
| Parallel Split (WP2) | + | + | + | + |
| Synchronization Merge (WP3) | - | + | + | + |
| Exclusive choice (WP4) | + | + | + | + |
| Simple Merge (WP5) | + | + | + | + |
| Multi-choice (WP6) | + | + | + | + |
| Synchronizing Merge (WP7) | - | - | +/- | + |
| Multi-merge (WP8) | - | + | + | + |
| Structured Discriminator (WP9) | +/- | + | + | + |
| Arbitrary cycles (WP10) | + | + | + | + |
| Implicit Termination (WP11) | + | + | + | - |
| MI without Synchronization (WP12) | + | + | + | + |
| MI with Priori Design-Time Knowledge (WP13) | - | + | + | + |
| MI with a Priori Run-Time Knowledge (WP14) | - | + | + | + |
| MI without a Priori Run-Time Knowledge (WP15) | - | - | - | + |
| Deferred Choice (WP16) | + | + | + | + |
| Interleaved Parallel Routing (WP17) | +/- | - | +/- | + |
| Milestone (WP18) | +/- | - | - | + |
| Cancel Task (WP19) | + | + | + | + |
| Cancel Case (WP20) | + | + | + | + |
| Structured loops (WP21) | + | + | + | + |
| Cancel Region (WP25) | + | + | + | + |
| Explicit Termination (WP43) | + | + | + | + |

Table 7.1: Evaluation of the expressiveness of EKF process language compared to AD, BPMN and YAWL

resembles a programming language, while different vendors provide tools for modeling. These tools have limited capabilities with respect to support any form of analysis (e.g. behavior verification, performance analysis, etc.). In other words, BPEL definitions are somewhere in-between the higher-level process models and fully-functional code. Hence, there are interesting efforts for translations relation to BPEL: first, a translation from a higher-level notation to BPEL (forward engineering in MDA) and second, translation from BPEL to a model to establish analysis (backward engineering in MDA).

Several attempts have been made to capture the formal semantics of BPEL. A comparative summary of formalizing BPEL can be found in [Wohed et al., 2003]. The first full formalization of the control flow as presented in the work of [Ouyang et al., 2005] is based on Petri-nets. Their work has resulted in a translation tool called BPEL2PNML and verification tool called WofBPEL[6]. Important is to notice that these verification tools only support a subset of the structured activities in BPEL. Alternative validation approach of BPEL is discussed in [Weidlich et al., 2007], where they evaluate BPEL based on $\pi$-calculus. With respect to the verification issues related communication aspect in

[6]Extension of the woflan verification tool, see: http://www.bpm.fit.qut.edu.au/projects/babel/tools.

BPEL, the work of [Martens, 2005] discusses how to verify the correctness of a collection of inter-communication BPEL processes, and similarly the work of [Fu et al., 2004] shows how to check the compatibility of two services with respect to their communication.

Various tools to generate BPEL code from a grapical representation are being developed by the Scientific, Open Source and Industry Communities. Tools such as Active BPEL[7], IBM WebSphere Choreographer[8] and the Oracle BPEL Process Designer[9] provide a graphical notation for BPEL. However, this notation directly reflects the code, so that the business oriented people still have to think in terms of BPEL constructs. More interesting is the work of White [2006] that discusses the transformation of BPMN into BPEL and the work of Koehler and Hauser [2004] on removing loops in the context of BPEL. Non of these publications provide an approach to transform some graphical process modeling language into BPEL, but merely present the problems, issues and challenges. More resent work on the development of an approach to transform Workflow-Nets into BPEL Aalst van der and Lassen [2005] and from a core sub-set of BPMN to BPEL Ouyang et al. [2006] have contributed in more rich approaches to translate higher-level process modeling language into BPEL code.

Other efforts focus on the backward engineering of BPEL, which come from the field of process mining, where event logs are used to derive conclusions of the process model [Aalst van der et al., 2004]. The work of Dongen van et al. [2005] has resulted in the PROM[10] tool and framework, which allows the analysis of different process definitions including BPEL. Another approach of backwards engineering is proposed by Brogi and Popescu [2006], where they present an approach to translate BPEL into YAWL.

Drawbacks associated with BPEL come directly from the limitations of the worfklow paradigm as discussed in chapter 2: limited flexibility, the atomicity of tasks, context tunneling and the mix-up of distribution and authorization. Case handling provides an alternative strategy to model and implement process, which need to be increasingly flexible to changes in the business environment [Aalst van der et al., 2005b]. Process models tend to become rather complex, due to the abundance of rules that can be described while analyzing only the business process [Vanthienen and Goedertier, 2005, Goedertier and Vanthienen, 2006]. This enables that the process becomes susceptible to design flaws, such that a substantial amount of effort is required for implementation and maintenance. This is mainly due to the fact that the business rules are maintained on a more fine-grained level compared to the process.

[7]see: http://www.activebpel.org
[8]see: http://www-306.ibm.com/software/websphere/
[9]see: http://www.oracle.com/technology/products/ias/bpel/index.html
[10]see: http://www.processmining.org

BPEL is mainly focused on the central coordination (orchestration) between participants and does not address problems where the transactions define the order of processing. For pure orchestration their remain challenges when managing processes, where coordination and synchronization between independent participants becomes increasingly complex [Aalst van der et al., 2005a]. The Web Service Choreography Definition Language (WS-CDL) [Barros et al., 2005a] is the latest development of W3C, toward an XML process language allowing choreography in the process. WS-CDL is not so much a replacement for BPEL, but can be seen as an standard on top of BPEL (in the web service stack) allowing to specify the interaction of global participating processes. Recent work of Barros et al. [2005b], Decker et al. [2007, 2008] describe patterns and language extensions to accomplish high-level interactions across processes. Nevertheless, at the time of writing there is only a working draft of WS-CDL available, so that adoption of this language depends of future success and implementations.

## 7.4 Further research

In this section we provide the insight in the further research and questions that arise from the shortcomings our work.

As our soloution only addresses the theoretical perspective of the problem, it is key for Everest that the solution is tanslated to an implementation. This requires an intergration of a BPEL engine into the EKF following a loosely coupled intergration approach. Implementing our proposed solution allows validation of the correctness and completeness, which is identified as a considerable obstacle as we only provided a pure theoretical approach.

The procedural process language like BPEL can be used to manage the generic process aspects independent of the business products and a specific organization. This implies that the strenght of BPEL is most applicable for the coarse-grained processes maintained by the EKF. The processes at a more fine-grained level become rather complex, when specified in a procedural langauge like BPEL. For such cases Everest aims at the separation of the rules from the flow to increase the maintainability and flexibility of the process. In this way the goals and choices are driven by rules rather than contained by the process itself. For Everest rules tend to be more subtitle to changes and reduce complexity issues as each rule can be seen as the smallest unit of change. Important insights for Everest can be found in the trade-off between a procedural flow (at the coarse-grained level) and a declarative flow (at a more fine-grained level). This level of abstraction allows generalizaiton in the process, where specialized flows aspects are implemented in terms of rules. An assessment of where to draw this line is therefore important for Everest to increase the understanding which aspects of the process need to be implemented in terms of procedural flow and which aspects as declarative rules. Managing the rules at a more fine-grained level is difficult, because the rule based flows become more complex when the number of rules (exception situations) increase. Managing this type of flows requires a different approach, techniques and standards. Furhter research should therfore adress the interoperability of the EKF on this fine-grained level of the process, which is not covered by BPEL.

# Chapter 8

# Conclusions

From the literature review we can conclude that there are several theoretical perspectives for enterprise modeling. Both the enterprise architecture and the model driven architecture are considered to be important approaches in the context of the Everest development life cycle for modeling and deployment of business applications. An extensive study of the Everest Knowledge Framework has resulted in a more clear understanding of the enterprise architecture and modeling dimensions adopted by Everest. The study of the domain, rule and process languages part of the Everest modeling dimensions, have resulted in a more clear understanding of the syntax of the supported languages. A study of BPEL has resulted into a more clear understanding of the key features of BPEL, including the underlying Web services technologies.

To determine how to increase the interoperability in the EKF by the adoption of BPEL we defined two partial questions: *'What are the issues and challenges to close the gap between the EKF process language and the BPEL?'* and *'How EKF process models can be transformed into BPEL code?'*. Although it is difficult to answer these questions from a theoretical perspective, we provided Everest with some interesting insights and results.

The transformation of the EKF process language into BPEL resulted in a more formal description of the syntax and semantics of the EKF process language and the contextual and conceptual differences between both languages. This is considered to be an important result for Everest as no previous attempt has been made by Everest to formalize the semantics of the EKF process language.

The contextual differences between the EKF process language and BPEL are identified when closing the gap of the meta-model elements. The conceptual differences between the EKF process language and BPEL are identified when closing the gap between the concepts of the EKF process language and BPEL. Various limitations arise from both the contextual and conceptual differences, which needs to be considered when translating the EKF process language into BPEL. First, the EKF process language lacks for supporting synchronization of parallel branches and does not allow multiple instances of tasks in the process. Second, BPEL only partially support the arbitrary cycle and structured discriminator patterns. This limitation introduces considerable complexities when translating the EKF process models into BPEL.

Transforming the EKF process models into BPEL code can be accomplished by: first, translate the domain; second, translate the technical spaces; third, decompose the (sub-)process into components and finally, translate the (sub-)process into BPEL code. Unless the difficulty to formally validate the completeness, we can say with considerable certainty that our approach can be applied to translate all EKF process models into BPEL. Unless the difficulty to formally validate the correctness, we can say with considerable confidence that our transformation result is correct.

On behalf of the results of this project we propose to either extend the expressiveness of the EKF process language or adopt a more formal modeling languages. The limitations of the EKF process modeling language need to be resolved in order to take fully advantage of the expressive power of BPEL, especially when automating the transformation of process models into BPEL code. Adopting a more formalized modeling language is recommended when automated transformation of the process design into a BPEL implementation is required. Automated generation of models fits the MDA strategy of Everest and seems useful when frequent changes to the process occur.

We recommend that a loosely coupled integration approach when integrating a BPEL engine into the EKF, as this approach promotes: interoperability, maintainability and scalability of the enterprise architecture of the EKF. This implies the adoption of Web service technologies in the EKF.

We recommend that further research covering the separation of a procedural flow at a coarse-grained level and rule based flow at a more fine-grained level of the process is valuable for Everest to promote flexible process management into the EKF.

# Bibliography

W.M.P. Aalst van der. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

W.M.P. Aalst van der. Workflow Verification: Finding Control-Flow Errors using Petri-net based techniques. *In W.M.P. van der Aalst, J. Desel, and A. Oberweis, editors, Business Process Management: Models, Techniques, and Empirical Studies*, 1806 of Lecture Notes in Computer Science:161–183, 2000.

W.M.P. Aalst van der and A.H.M. Hofstede ter. Workflow Patterns: On the Expressive Power of (Petri-net-based) Workflow Languages. *Wor. In K. Jensen, editor, Proceedings of the Fourth Workshop on the Practical Use of Coloured Petri Nets and CPN Tools (CPN 2002)*, 560 of DAIMI:1–20, August 2002.

W.M.P. Aalst van der and A.H.M. Hofstede ter. YAWL: Yet Another Workflow Language (revised version). *QUT Technical report, FIT-TR-2003-04*, 2003. URL http://www.citi.qut.edu.au/pubs/technical/yawlrevtech.pdf.

W.M.P. Aalst van der and K.B. Lassen. Translating Workflow-nets to BPEL4WS. *BETA Working Paper Series*, 2005.

W.M.P. Aalst van der and K. van Hee. *Workflow Management*. MIT Press, 2004. ISBN 0-262-72046-9.

W.M.P. Aalst van der, A.H.M. Hofstede ter, B. Kiepuszewski, and Barros A.P. Workfow Patterns. *Distributed and Parallel Databases*, 14(3):5–51, June 2003.

W.M.P. Aalst van der, A.J.M.M. Weijters, and editors. Process mining. *Special Issue of Computers in Industry*, 53(3), 2004.

W.M.P. Aalst van der, M. Dumas, A.H.M. Hofstede ter, N. Russell, H.M.W. Verbeek, and P. Wohed. Life After BPEL? *In Proc. of the 2nd Int. Workshop on Web Services and Formal Methods (WS-FM)*, 3670 of LNCS:35–50, 2005a.

W.M.P. Aalst van der, M. Weske, and D. Grunbauer. Case handling: a new paradigm for business process support. *Data Knowl. Eng.*, 53(2):129–162, 2005b.

D.H. Akehurst. Validating bpel specifications using ocl. *Technical report.*, August 2004.

C. Atkinson and T. Kuhne. Model-driven development: a metamodeling foundation. *Software, IEEE, Publication Date: Sept.-Oct. 2003*, 20(5):36– 41, 2003.

M. Bajec and M. Krisper. A methodology and tool support for managing business rules in organisations. *Information Systems*, 30(6), September 2005.

M. Bajec, R. Rubnik, and M. Kisper. Using Business Rules Technologies To Bridge The Gap Between Business and Business Applications. *Information Technology for Business Management (G Rechnu, Ed)*, Proceedings of the IFIP 16th World Computer Congress 2000:77–85, 2000.

A. Barros, M. Dumas, and P. Oaks. A Critical Overview of the Web Service Choreography Description Level (WS-CDL). *BPTrends Newsletter*, 3(3), March 2005a.

A. Barros, A. Hoftede ter, and M. Dumas. Service Interaction Patterns. *BPM 2005*, LNCS 3649:302–318, 2005b.

BEA, Microsoft, IBM, SAP, and Siebel. Business Process Execution Language for Web Services (Version 1.1). *IBM Developerworks*, 2003.

B. Bordbar and A. Staikopoulos. Bridging Technical Spaces With A Metamodel Refinement Approach. *Electronic Notes in Theoretical Computer Science*, 2005.

A. Brogi and R. Popescu. From BPEL Processes to YAWL Workflows. *Springer*, 4184 Lecture Notes in Computer Science:107–122, December 2006.

D. Carlson. Modeling XML Vocabularies with UML: Part I. *O'Reilly*, August 2001a. URL http://www.xml.com/pub/a/2001/08/22/uml.html.

D. Carlson. Modeling XML Vocabularies with UML: Part II. *O'Reilly*, September 2001b. URL http://www.xml.com/pub/a/2001/09/19/uml.html.

D. Carlson. Modeling XML Vocabularies with UML: Part III. *O'Reilly*, October 2001c. URL http://www.xml.com/pub/a/2001/10/10/uml.html.

U. Dayal, A.P. Buchmann, and D.R. McCharty. Rules are Objects too: A Knowledge Model for an Active, Object-Oriented Database Management System. *Springer Berlin*, Advances in Object-Oriented Database Systems (Ed. K.R. Dittrich):29–143, 1998.

T. Debevoise. *Business Process Management With a Business Rules Approach: Implementing the Service Oriented Architecture*. Business Knowledge Architects, 2005. ISBN 0-9769048-0-2.

D. Decker, O. Kopp, F. Leymann, and M. Weske. Modeling Service Choreographies using BPMN and BPEL4Chor. *Proceedings of the 20th International Conference on Advanced Information Systems Engineering (CAiSE)*, June 2008.

G. Decker and F. Puhlmann. Extending BPMN for Modeling Complex Choreographies. *Proceedings of the 15th International Conference on Cooperative Information Systems (CoopIS)*, 4803 of LNCS:24–40, November 2007.

G. Decker, O. Kopp, F. Leymann, and M. Weske. BPEL4Chor: Extending BPEL for Modeling Choreographies. *In Proceedings International Conference on Web Services (ICWS)*, 2007.

R. M. Dijkman, M. Dumas, and C. Quyang. Formal semantics and automated analysis of BPMN process models. *Preprint 7115, Queensland University of Technology,Brisbane, Australia*, 2007.

B. Dongen van, A.K.A. Medeiros de, H.M.W. Verbeek, A.J.M.M. Weijters, and W.M.P. Aalst van der. The ProM framework: A New Era in Process Mining Tool Support. *In G. Ciardo and P. Darondeau, editors, Application and Theory of Petri Nets 2005*, 3536 of Lecture Notes in Computer Science:444–454, 2005.

EnixConsluting. Business rules are from Mars & processes from Venus. *Technical Report*, 2005. URL http://www.bpmlabs.net/images/paper/Rules.pdf.

X. Fu, T. Bultan, and J. Su. Analysis of interacting BPEL Web services. *In Proceedings of the 13th International Conference on World Wide Web*, pages 621–630, 2004.

D. Gasevic, A. Giurca, S. Lukichev, and G. Wagner. Rule-Based Modeling of Semantic Web Services. *Proceedings of 2nd International Workshop on Semantic Web Enabled Software Engineering (SWESE 2006)*, pages 5–9, November 2006. URL http://km.aifb.uni-karlsruhe.de/ws/swese2006/final/gasevic_short.pdf.

S. Goedertier and J. Vanthienen. Compliant and Flexible Business Processes with Business Rules. *In 7th Workshop on Business Process Modeling*, 2006.

M. Harvey. *Essentials Business Process Modeling*. O'Reilly, 2005. ISBN 0-596-00843-0.

H. Herbst. The specification of business rules: a comparison of selected methodologies. *Life Cycle, Amsteredam et al., Elsevier*, pages 29–46, 1994.

H. Hermans. 5GL springlevend (dutch). *Computable*, Feburari 2007. URL http://www.computable.nl/artikel.jsp?rubriek=1509029&id=1855254.

D. Hollingworth. The Workflow Reference Model. *Workflow Management Coalition*, January 1995. URL http://www.wfmc.org/standards/docs/tc003v11.pdf.

IBM and SAP. WS-BPEL Extension for People. *Technical Report*, July 2005.

IBM, BEA, and Microsoft. Business Process Execution Language for Web Services (Version 1.0). *Technical Report*, July 2002.

K. Jensen. Coloured Petri Nets: Analysis Methods and Practical Use. *Springer-Verlag*, Vol. 1: Basic Concepts Monographs in Theoretical Computer Science, 1997.

Jumper. Making BPEL Transformations Dynamic. *Jumper Newsletter*, October 2006. URL http://www.jumpernetworks.com/Making_BPEL_Transformations_Dynamic.pdf.

A. Kleppe. *The Model Driven Architecture (MDA): Practice and Promise*. Addison-Wesley, 2003. ISBN ISBN 0-321-19442-X.

J. Koehler and R. Hauser. Untangling unstructured cyclic flows: A solution based on continuations. *In Proceedings of OTM Confederated International Conferences CoopIS, DOA, and ODBASE 2004*, pages 121–138, 2004.

G. Kontonya and I. Sommerville. *Requirements Engineering*. Wiley, 2002. ISBN ISBN 0-471-9720-8.

K. Kriz, J. Gehrke, D. Kriz, J. Vanthienen, C. Mues, G. Wets, and K. Delaere. A tool-supported approach to inter-tabular verification. *Expert Systems with Applications 15*, pages 277–258, 1998.

U. Kuster and B. Konig-Ries. Dynamic binding for BPEL processes: A Lightweight Approach to Integrate Semantics into Web Services. *Service-Oriented Computing ICSOC 2006*, 4652 Lecture Notes in Computer Science:116–127, September 2007.

F. Leymann. Web Services Flow Language (WSFL). *IBM Software Group*, 2001. URL http://www-306.ibm.com/software/solutions/Webservices/pdf/WSFL.pdf.

P. Loucopoulos. From Information Modelling to Enterprise Modelling. *in Information-Systems Engineering: State of the Art and Research Themes, (Ed: S. Brinkkemper, E. Lindencrona,A. Solvberg)*, pages 67–78, 2000.

P. Loucopoulos, B. Theodoulidis, and D. Pantazis. Business Rules Modelling: Conceptual Modelling and Object Oriented Specifications. *Proceedings of the IFIP TC8/WG8.1 Working Conference*, pages 323–342, 1991.

A. Martens. Analyzing Web Service Based Business Processes. *In M. Cerioli, editor, Proceedings of the 8th International Conference on Fundamental Approaches to Software Engineering (FASE 2005)*, 3442 of Lecture Notes in Computer Science:19–33, 2005.

S.A.R.L. Martin. The role of business rules in SOA. *Technical Report*, 2006. URL http://www.visual-rules.com/pdf_en/whitepaper-business-rules-soa.pdf.

S. Mellor. *Principles of Model Driven Architecture*. Addison-Wesley Professional, 2004. ISBN ISBN 0-201-78891-8.

B. M. Michelson. Event-Driven Architecture Overview. *Patricia Seybold Group*, Febuary 2006.

T. Morgan. *Business Rules and Information Systems: Aligning IT with Business Goals*. Addison-Wesley, 2002. ISBN 0-201-74391-4.

M. Muehlen zur. *Workflow Based Process Controlling*. Logos Verslag Berlin, 2004. ISBN 3-8-325-0388-9.

N.A. Mulyar. Pattern-based Evaluation of Oracle-BPEL (v.10.1.2). *Technical report, Center Report BPM-05-24*, 2005.

A. Newell. The knowledge level. *Artificial Intelligence*, 18:87–127, 1982.

OASIS. Web Services Business Process Execution Language (Version 2.0). *Public Review Draft*, August 2006.

OMG. Unified Modeling Language (UML) 2.0. *Object Management Group*, 2005. URL http://www.omg.com/uml/.

OMG. Production Rule Representation (PRR) (proposal). *Draft Response to OMG RFP br/2003-09-03*, 6 2006.

OMG. MDA Guide (Version 1.0.1). *Document Number: omg/2003-06-01*, June 2003. URL http://www.omg.org/docs/omg/03-06-01.pdf.

B. Orriens, J. Yang, and M.P. Papazoglou. A Framework for Business Rule Driven Service Composition. *In Proceedings of the Fourth International Workshop on Conceptual Modeling Approaches for e-Business Dealing with Business Volatility*, 2003.

C. Ouyang, W.M.P. van der Aalst, S. Breutel, M. Dumas, A.H.M. Hofstede ter, and H.M.W. Verbeek. Formal Semantics and Analysis of Control Flow in WS-BPEL. *BPM Center Report BPM-05-15*, 2005.

C. Ouyang, W.M.P. Aalst van der, M. Dumas, and A.H.M. Hofstede ter. Translating BPMN to BPEL. *BPM Center Report BPM-06-02*, 2006.

F. Puhlmann and M. Weske. Using the $\pi$-calculus for Formalizing Workflow Patterns. *In BPM 2005*, 3649 of Lecture Notes in Computer Science:153–168, 2005.

H.A. Reijers and S. Limam Mansar. Best Practices in Business Process Redesign: Validation of a Redesign Framework. *Computers in Industry*, 56(5):457–471, 2005.

D. Rosca, S. Greenspan, C. Wild, H. Reubenstein, K. Maly, and M. Felowitz. Application of a decision support mechanism to the business rules life cycle. *Proceedings of the 10th Knowledge-Based Software Engineering Conference*, pages 114–121, November 1995.

F. Rosenberg and S. Dustdar. Business Rules Integration in BPEL - A Service-Oriented Approach. *In: CEC.*, pages 476–479, 2005a.

F. Rosenberg and S. Dustdar. Towards a Distributed Service-Oriented Business Rules System. *Proceedings of the Third European Conference on Web Services (ECOWS05)*, 2005b.

R.G. Ross. Business Rules Manifesto (version 2.0). *The principles of Rule Independence*, 2003.

R.G. Ross. Principles of the business rules approach. *Data Base Systems: Design, Implementation and Management*, 2000.

N. Russell, W.M.P. Aalst van der, A.H.M. Hofstede ter, and P. Wohed. On the suitability of UML 2.0 Activity Diagrams for business process modelling. *In to appear in Proc. of The 3rd Asia-Pacific Conf, on Conceptual Modelling (APCCM 2006)*, January 2006.

R. Schmidt. Web services based execution of business rules. *In Proceedings of the International Workshop on Rule Markup Languages for Business Rules on the Semantic Web*, 2002.

G. Schreiber, H. Akkermans, A. Anjewierden, R. Hoog de, N. Shadbolt, W. Velde van de, and B. Wielinga. *Knowledge Engineering and Management: The CommonKADS Methodology.* IT Press, 2000. ISBN 0-262-19300-0.

A.K. Tanaka. On Conceptual Design of Active Databases. *PhD Thesis*, 1992.

S. Thatte. XLANG Web Services for Business Process Design. *Technical Paper*, 2001.

J. Vanthienen. Knowledge Acquisition and Validation Using a Decision Table Engineering Workbench. *Proceedings of the World Congress on Expert Systems*, pages 16–19, 1991.

J. Vanthienen and S. Goedertier. Rule-based business process modeling and execution. *In: Proceedings of the IEEE EDOC Workshop on Vocabularies Ontologies and Rules for The Enterprise (VORTE 2005)*, CTIT Workshop Proceeding Series (ISSN 0929-0672), 2005.

W3C. Rule Interchange Format Working Group Charter (RIF). *W3C Charter*, 26(3), 2005. URL http://www.w3.org/2005/rules/wg/charter.html.

G. Wagner. How to design a general rule markup language? *In Workshop XML Technologien fuer das Semantic Web (XSW), Berlin*, June 2002.

M. Weidlich, G. Decker, and M. Weske. Efficient Analysis of BPEL 2.0 Processes using $\pi$-calculus. *Proceedings of the IEEE Asia-Pacific Services Computing Conference (APSCC), Tsukuba Science City*, December 2007.

WFMC. Workflow Management Application Programming Interface Specification (Version 2.0). *Technical report*, July 1998.

WfMC, FileNet, ILOG, and W4. Business Processes and Business Rules: Business Agility Becomes Real. *Workflow Handbook 2003*, 2003. URL http://www.ilog.com/solutions/business/bpm/WfMC_article_BPM_and_BRE.pdf.

S. White. Business Process Modeling Notation (BPMN) (version 1.0). *Business Process Management Initiative*, May 2004.

S.A. White. Using BPMN to model a BPEL process. *BPTrends*, 3(3):1–18, October 2006. URL http://www.bptrends.com/.

J. Widom and S. Ceri. Active Database Systems: Triggers and Rules For Advanced Database Processing. *Morgan Kaufmann*, 1996.

L. Wilkes and R. Veryard. Service-Oriented Architecture: Considerations for Agile Systems. *Microsoft Architects Journal*, pages 11–23, April 2004.

P. Wohed, W.M.P. van der Aalst, M. Dumas, and A.H.M. Hofstede ter. Analysis of web services composition languages: The case of BPEL4WS. *In Proceedings of 22nd International Conference on Conceptual Modeling (ER 2003)*, volume 2813, of Lecture Notes in Computer Science:200–215, 2003.

P. Wohed, W.M.P. Aalst van der, M. Dumas, A.H.M. Hofstede ter, and N. Russell. Pattern-based analysis of the control-flow perspective of UML Activity Diagrams. *ER 2005*, pages 63–78, 2005a.

P. Wohed, W.M.P. Aalst van der, M. Dumas, A.H.M. Hofstede ter, and N. Russell. Pattern-based analysis of BPMN. *Queensland University of Technology*, 2005b.

P. Wohed, W.M.P. Aalst van der, M. Dumas, A.H.M Hofstede ter, and N. Russell. On the suitability of BPMN for Business Process Modeling. *In Proceedings 4th International Conference on Business Process Management (BPM 2006)*, 2006.

J.A. Zachman. A Framework for Information Systems Architecture. *IBM Systems Journal*, 26(3), 1987.

J. M. Zaha, M. Dumas, A. Hofstede ter, A. Barros, and G. Decker. Service Interaction Modeling: Bridging Global and Local Views. *In Proceedings 10th IEEE International EDOC Conference (EDOC 2006)*, October 2006.

# Appendix A

# One Step Refinement Approach Explained

In this thesis we use the *"one step refinement"* approach which aims at bridging the gap between between the technical spaces and domains of languages. Both the technical spaces and domain should be considered when mapping the two meta-models, where the technical spaces focuses on the technology issues and implementation issues and the domain on the conceptual differences, which includes the semantics of the model execution.

## A.1 Introduction

The idea of one step refinement is to bridge two technical spaces by considering a source and destination. In order to map two technical spaces, one needs to identify and match their corresponding meta-model elements and supporting characteristics. This is not an easy task, because the meta-models define a complex structure for which the behavior is defined by its conceptual interpretation. Therefore each match on the technical spaces must be evaluated by determine the semantical equivalence of the mapping. Still there exist elements in the source language for which no direct mapping between the elements can be identified, due to the syntactical differences between the languages. One step refinement handles such cases by decomposing the source element, that can not directly be mapped to the destination. This can be done by introducing it a new concept in the destination language. The following definitions give a more formal representation of the one step refinement approach.

## A.2 Description

Assuming that the aim is to define a model transformation from a technical space modeled via a meta-model M (EKF process) into a technical space modeled via a meta-model N (BPEL) (see Figure A.1). At this example, emphasis is placed upon one-way mappings from M to N, where the meta-model of the source M is assumed to be more expressive or richer than the target meta-model N, in the sense that there is a considerable number of meta-model elements of M, which cannot be directly mapped into meta-model elements of N.

In some cases, it might be possible to map some of the model elements of M into N as depicted by the $\psi_0$ mapping. Now, consider a model element $\beta_1$ of M that cannot be directly mapped into any model elements of N. However, suppose that it is possible to construct $\beta_1$ via model elements of $N := N_0$. This leads to one step refinement $M_1 := N_1 \oplus \alpha_1$ such that $\beta_1$ can be mapped to $\alpha_1$. The process can be repeated until an extension $N_k$ (after $k$ stages) is created, such that all model elements of $M$ can be mapped successfully and meaningfully into model elements of $N_k$.

As a result of the refinement, the technical spaces of $N_k$ and $M$ are now close enough to be mapped completely, as depicted by $\Psi_k$. Now, if all step refinements from $N$ to $M$ are decomposable, then it is possible to define model transformations $\Psi_i$ from $M_i$ to $M_{i-1}(1 \leq i \leq k)$ such that the mapping $\varphi = \psi_0 \circ \psi_1 \circ \psi_2 ... \psi_{k-1}, \psi_k$ maps M to N.

**Step 1:** Assume that a source meta-model $M$ and a destination meta-model $N$ and $N$ needs to be mapped to $M$, under the condition that the meta-model of $N$ is more expressive than the meta-model of $M$. The total mapping can be defined by a finite set of mappings $k$ between $M \; \psi \; N$, which can be expressed by $\forall_{(1 \leq i \leq k)}(M \; \psi_i \; N)$.

**Step 2:** A **direct mapping** $\alpha \; \psi_i \; \beta$ is a mapping where the a source element $\alpha_i \in M$ can be directly be mapped to a destination element $\beta_i \in N$.

**Step 3:** If no direct mapping $\alpha \; \psi_i \; \beta$ then a **One step refinement** $N \oplus \alpha_i$ must be performed by introducing $\alpha_i$ to $N$. Then, we say that $N \oplus_i \alpha$ is **decomposable**, if we can define a transformation pattern (in terms of a function) $\oplus_i$ from $\varphi := M \oplus_i \alpha$ to $N$, without losing essential information as described by the $\alpha \in N$ that is being mapped.

The difficulty of the meta-model transformation approach is clearly the third step, to find the transformation pattern for the concepts, which can not directly be mapped onto the destination concepts. For this step a more detailed study is required to find matching

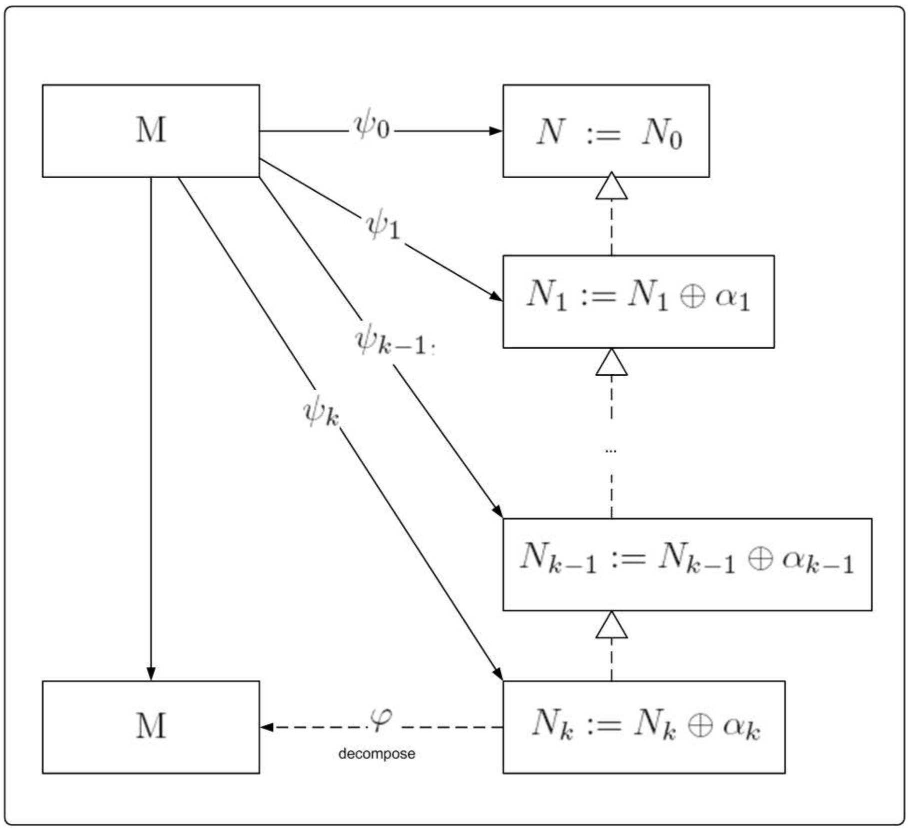

M
$\psi_0$
$N := N_0$
$\psi_1$
$N_1 := N_1 \oplus \alpha_1$
$\psi_{k-1}$
$\psi_k$
...
$N_{k-1} := N_{k-1} \oplus \alpha_{k-1}$
M
$\varphi$
decompose
$N_k := N_k \oplus \alpha_k$

Figure A.1: One Step Refinement Approach.

pattern that exist in both languages, for which a pattern can be defined as: *'a family of solution of a class of recurring design problems'*. In our study we use an abstraction of these concepts to be added as meta-model elements in the target langauge.

# Appendix B

# Domain transformation

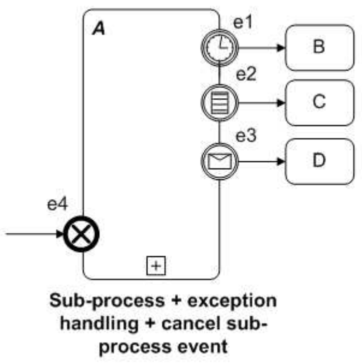
The aim of the domain transformation is to determine for which language elements supported by the EKF process language a BPEL equivalent exists. This all the elements from the meta-model of the EKF process language: (sub-)process, task, event types, flows, decision-points, channel, role and data. Important is to determine the main differences in how both language elements are threaded by the underlying implementation.

## B.1 Process, Sub-process and Task

The *process and sub-process* in the EKF process language specify a hierarchical grouping of correlated tasks sharing the same exception handling and trigger events (see Figure B.1). In BPEL this can be achieved through the scope activity. Complexity issues arise as the scope requires one entry and one exit point, which is clearly violated by the process and sub-process elements of the EKF process language. A solutions to the problem requires that the process and sub-process in the EKF process language must be translated into respectively an abstract (sub-)process and abstract process element for which the BPEL scope requirement must be satisfied (see Figure B.3). *Exceptions* are translated into an event handler in the scope activity and the *cancel sub-process* event are translated into a combination of an event and fault handler. Notice that the process element in the EKF process language is translated into a BPEL process and sub-process into the scope activity.

The *tasks* in the EKF process language can directly be mapped onto the invoke activity in BPEL (see Figure B.2 and B.3). In the EKF the actual execution of the task directly relates to the execution of a certain channel, for which a certain role is required. The channel prescribes the operation, which must be performed in order to pursue a certain

**EKF element** | **BPEL code**

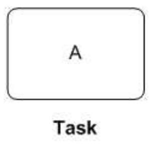


```
<scope name="A">
 <faultHandler>
  <catch faultName="fe4">
   <empty/>
  </catch>
 </faultHandler>
 <eventHandler>
  <onAlarm name="e1">
   <!-- code for B -->
  </onAlarm>
  <onMessage operation="e2">
   <!-- code for C -->
  </onMessage>
  <onMessage operation="e3">
   <!-- code for D -->
  </onMessage>
  <onMessage operation="e4">
   <throw faultName="fe4"/>
  </onMessage>
 </eventHandler>
 <!-- code for A -->
</scope>
```

Figure B.1: Translate sub-process element.

goal of the task. The role and channel in the EKF relates to a *partnerlink* and *porttype* of the BPEL invoke activity. The channel specifies the operation and input and output variables can directly be mapped onto respectively the operation, input-variable and output-variable of the BPEL invoke activity. A more detailed discussion on translating the channel and role into BPEL is presented in section B.4.

**EKF element** | **BPEL code**



```
<invoke operation="A" parnerLink="role1"
 portType="channel1" inputVariable="paramin"
 outputVariable="paramout"/>
```

Figure B.2: Translate task element.

## B.2 Events

The EKF process language support start point events, intermediate events and end point events. We continue to describe the mapping of each event type to the corresponding BPEL equivalent. An overview of the meta-model mapping of the EKF events is presented in figure B.6.

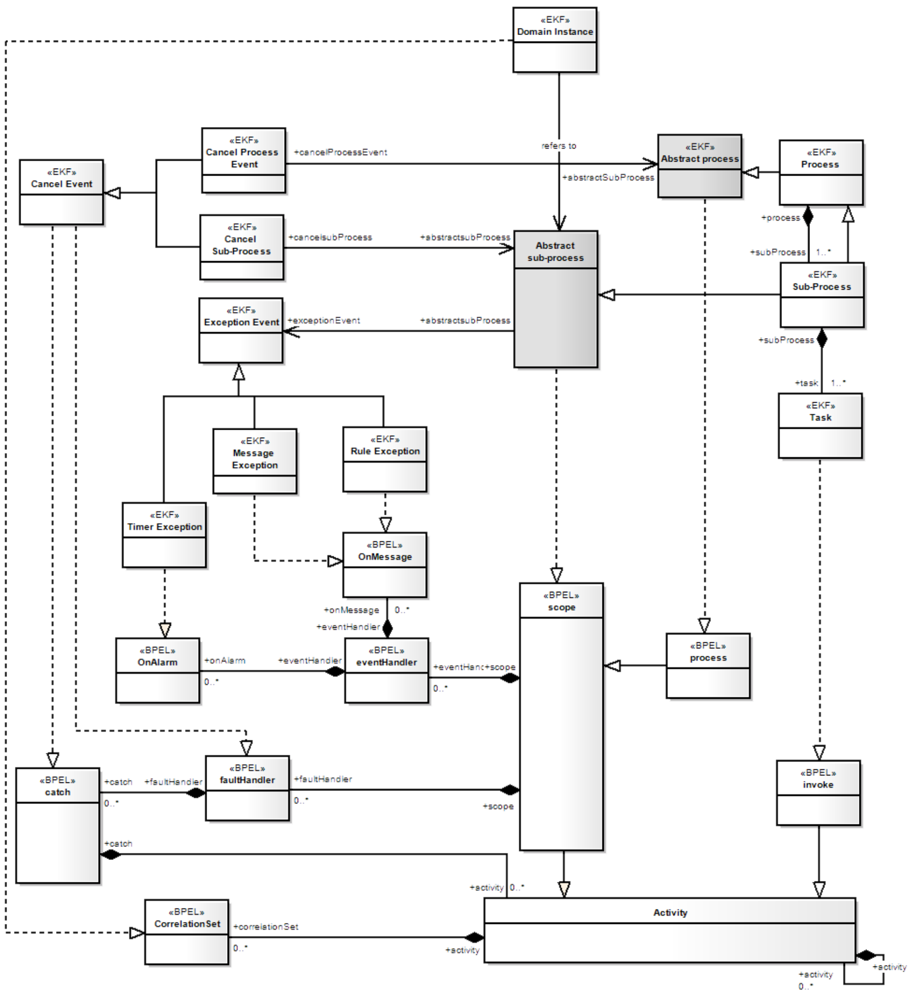


Figure B.3: Meta-model mapping of process, sub-process and task.

The EKF process requires minimal one *start event*. The start of the process results in the creation of an instance in the process and can directly be mapped onto the receive activity in BPEL, for which the property *createInstance* is set to *yes* (see start event in Figure B.4).

The *process language* allows multiple start events. Translating the multiple start events into BPEL requires the use of a pick activity for which the occurrence of the first *onMessage* or *onAlarm* results in the creation of the process instance. This also allows that the process is activated through either the expiration of a timer, violation of a rule or the reception of a message (see multiple start events in Figure B.4). The rule events are not directly supported by BPEL, but could alternatively be implemented as a message event, which is triggered by an external rule engine.

The *trigger event* is more or less an interrupt, which can occur only if a certain condition is satisfied. After the occurrence of a trigger event and if the condition is satisfied

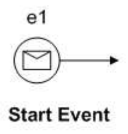
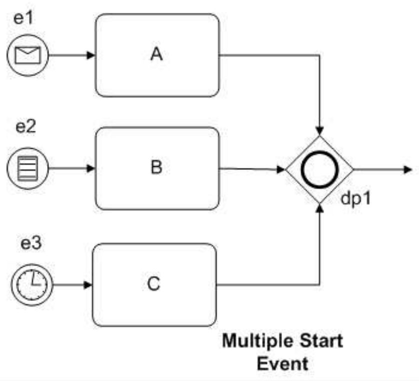
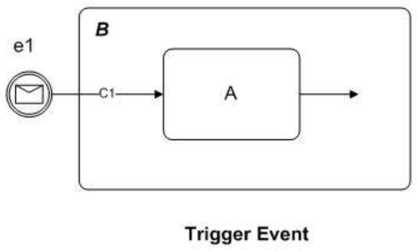
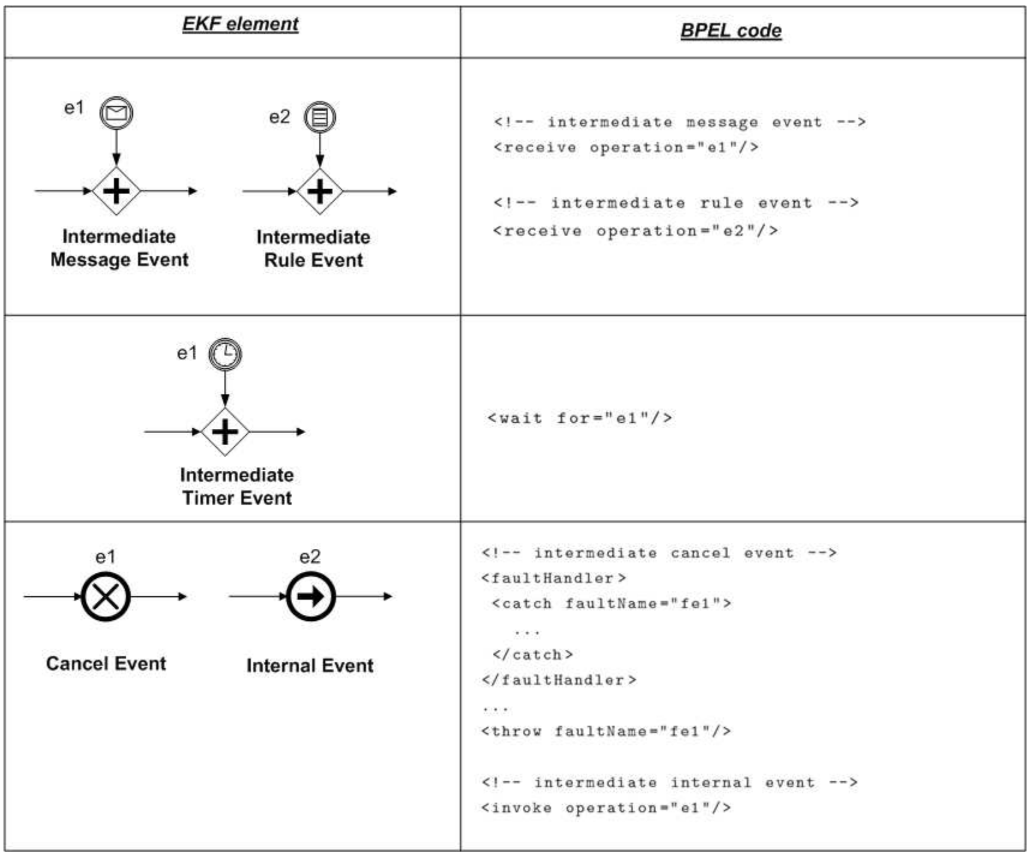
| ***EKF element*** | ***BPEL code*** |
|---|---|
|  | `<receive operation="e1" createInstance="yes"/>` |
|  e1<br>A<br>e2<br>B<br>dp1<br>e3<br>C<br>Multiple Start Event | `<pick>`<br> `<onMessage operation="e1" createInstance="yes">`<br>  `<!-- code for A -->`<br> `</onMessage>`<br> `<onMessage operation="e2" createInstance="yes">`<br>  `<!-- code or B -->`<br> `</onMessage>`<br> `<onAlarm name="e1" createInstance="yes">`<br>  `<!-- code for C -->`<br> `</onAlarm>`<br>`</pick>` |
|  | `<scope name="B">`<br> `<faultHandler>`<br>  `<catch faultName="fe1">`<br>   `<!-- code for A -->`<br>  `</catch>`<br> `</faultHandler>`<br> `<eventHanlder>`<br>  `<onMessage operation="e1">`<br>   `<switch>`<br>   `<case condition="C1">`<br>    `<sequence>`<br>     `<invoke operation="A"/>`<br>     `<throw faultName="fe1"/>`<br>    `</sequence>`<br>   `</case>`<br>   `<otherwise>`<br>    `<empty/>`<br>   `</otherwise>`<br>  `</switch>`<br>  `</onMessage>`<br> `<!-- event handlers for A -->`<br> `</eventHandler>`<br> `<!-- code for B -->`<br>`</scope>` |

FIGURE B.4: Translate start point events element.

then all instances in the (sub-)process are aborted and processing is continued from an alternative point in the process (event forward). In case the condition is not satisfied, no additional actions are preformed. In BPEL the trigger event can be implemented as an *event handler* throwing a *fault*. The occurrence of the fault results in the abortion of all the active instances within a scope activity, such that the processing is performed by the corresponding *catch* of the *fault handler* (see trigger event in Figure B.4).

The *intermediate message and timer events* can directly be mapped onto respectively the BPEL *receive* activity (see intermediate message event in Figure B.5) and *wait* activity (see intermediate timer event in Figure B.5). The receive activity waits for the



```
<!-- intermediate message event -->
<receive operation="e1"/>

<!-- intermediate rule event -->
<receive operation="e2"/>
```

```
<wait for="e1"/>
```

```
<!-- intermediate cancel event -->
<faultHandler>
 <catch faultName="fe1">
   ...
 </catch>
</faultHandler>
...
<throw faultName="fe1"/>

<!-- intermediate internal event -->
<invoke operation="e1"/>
```

FIGURE B.5: Translate intermediate events element.

reception of a message from an external system or process and the wait activity waits for the expiration of a timer constant. The *intermediate rule events* are not directly supported by BPEL, such that integration of an external rule engine in BPEL is required to accomplish rue based events.

The *end event* in the EKF process specifies the completion of the process, such that no further actions can be performed. The end event can be mapped onto an *invoke* activity, under the condition that this activity is the last activity in the process (see end event in Figure B.6).

The *termination event* is directly supported by BPEL through the *terminate* activity (see termination event in Figure B.6).

The NOP event is directly supported by the BPEL through the *empty* activity (see nop event in Figure B.6).

The *notification event* interacts with external systems or processes by means of a system-to-system interaction. In the EKF the system-to-system interaction is implemented in terms of a channel in the connectivity layer. Translating the notification event into BPEL implies that the channel is exposed as Web-services. Therefore the notification

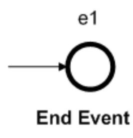
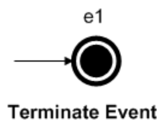
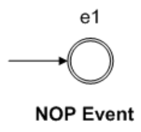
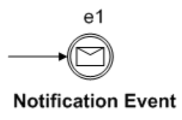
| ***EKF element*** | ***BPEL code*** |
|---|---|
|  | `<!-- for asynchronous processes -->`<br>`<invoke operation="e1"/>`<br><br>`<!-- for synchronous processes -->`<br>`<reply operation="e1"/>` |
|  | `<terminate name="e1"/>` |
|  | `<empty/>` |
|  | `<invoke operation="e1"/>` |

FIGURE B.6: Translate end point events element.

message can directly be mapped onto the invoke activity in BPEL (see notification event in Figure B.6).

## B.3 Flow and Decision-points

The flow and decision-points link the set of elements as presented in previous sections to define the control flow in the process. For both the flow and decision-points no direct mappings can be identified in BPEL. This is mainly caused by the fact the EKF and BPEL process language belong to two fundamentally different classes of languages for these elements. The EKF process language is based on a block-oriented language, while the EKF process language is based on a graph-oriented language. The block-oriented structure of BPEL is directly inherited by the nature of XML and limits the basic activities to have one entry point and one exist point. As a result of this observation, we define the flow and decision-points elements of the EKF process language to be fundamentally different from the activity hierarchy adopted by BPEL.

## B.4 Channel and Roles

In the EKF an channel describes the services required for the actual execution of the task at a more fine-grained level of the process. BPEL uses a similar concept, which

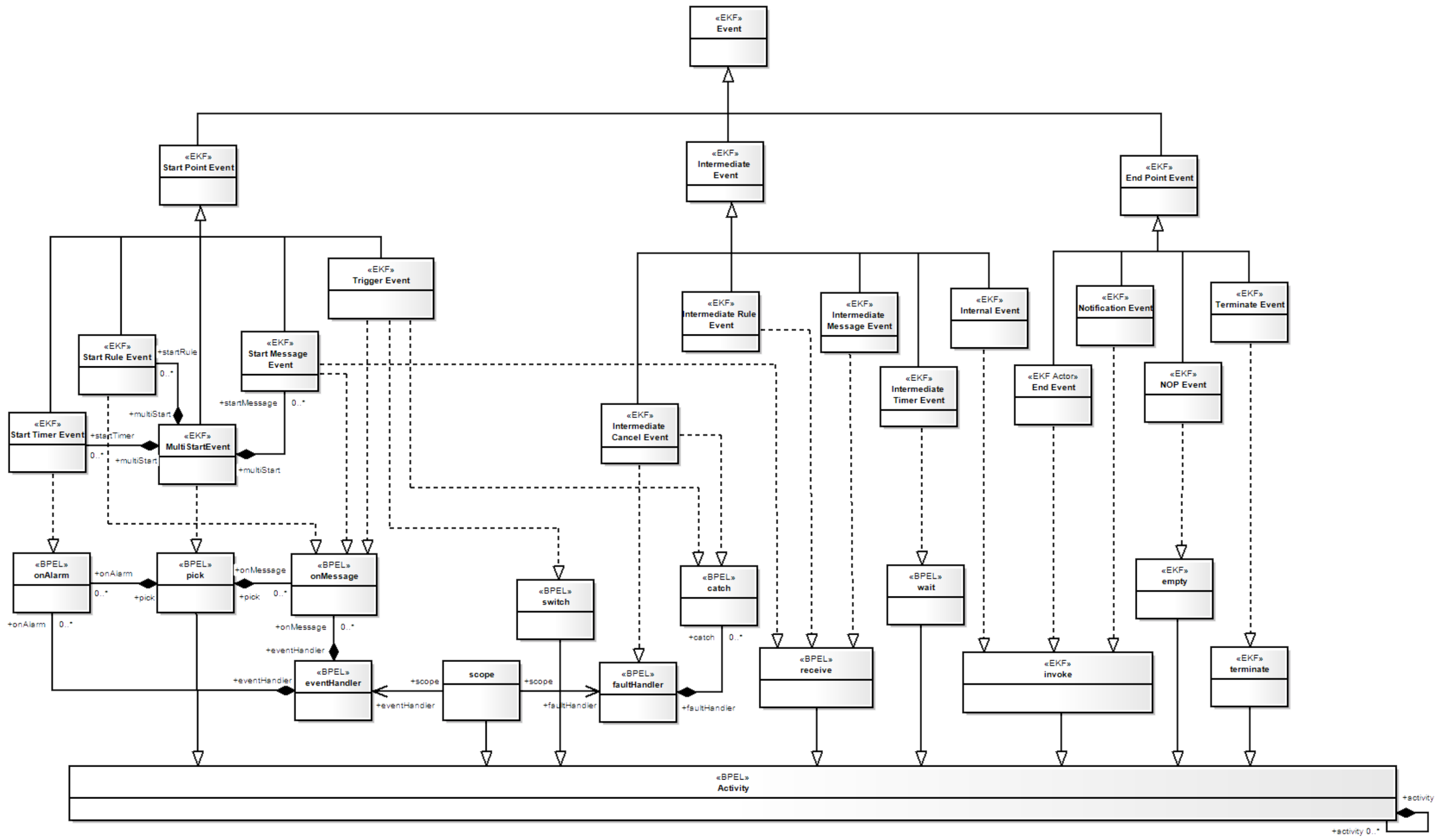


Figure B.7: Meta-model mapping of events.

is completely based around Web services. BPEL requires a WSDL definition, which specifies the Web service operations used to reference to the partner activities (e.g. invoke, receive and reply). In BPEL a partner link is used to glue the WSDL operations and the BPEL activities through roles. A partner link in the BPEL references to the partner link type in the WSDL definition, where the WSDL definition specifies the the actual service details, such as role and port type. In the WSDL definition the details of the Web service, includes the mappings between the operations and the partner link types. The partner link type could be composed of multiple roles as BPEL allows both a-synchronous and synchronous communications for the partner activities. In listing B.1 we give an example of the concept of partnerlinks, partnerlink types and port types in the BPEL and underlying WSDL definitions.

```
<-- Example of partner links in the BPEL definition -->
<partnerLinks>
 <partnerLink name="ChannelName"
  partnerLinkType="WSDLpartnerLinkType"
  myRole="Role" partnerRole="Role"/>
</partnerLinks>
<-- Example of partner link types in the WSDL definition -->
<partnerLinkType name="Channel">
 <role name="RequestRole">
  <portType name="WSDLportType"/>
 </role>
 <role name="ReplyRole">
  <portType name="WSDLportType"/>
 </role>
</partnerLink>
<-- Example of operation in the WSDL definition -->
<portType name="WSDLoperation">
 <operation name="WSDLOperation">
  <input message="WSDLmessage"/>
  <output message="WSDLmessage"/>
 </operation>
</portType>
```

Listing B.1: Mapping EKF channels to BPEL partnerLinks

From these observation we can further discuss the transformation of the channel and role into the partner links and port types of BPEL. The EKF channel can directly be mapped to a WSDL definition in BPEL as each channel operation should be exposed as a Web service. BPEL allows both synchronous an a-synchronous configurations for Web-services, therefore it is required that the role in the EKF is specialized into a consumer and provider role. The channel operation in the EKF directly corresponds to a WSDL operation, for which the input and output variables are defined in terms of an WSDL message. The WSDL message are defined in terms of XSD elements, for which a more detailed description of transforming the data of EKF into XSD is presented in the following section.

From these observations we can provide an extensible meta-model presenting the direct mappings between the EKF and BPEL + WSDL (see Figure B.8).

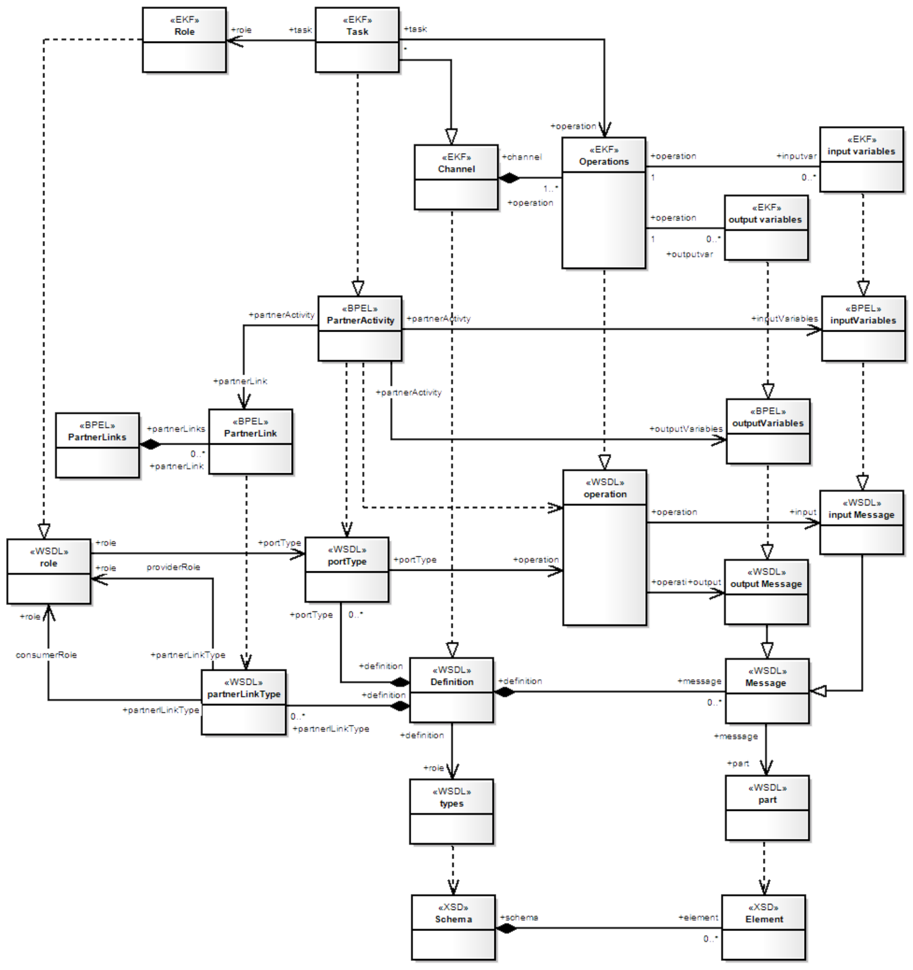


Figure B.8: Meta-model mapping of EKF and WSDL.

## B.5 Data

In the EKF the domain model only specifies the format and constraints of the data. In BPEL the data is defined in terms of a WSDL messages, which is composed of XSD elements. Closing the gap between the data perspective of the EKF and BPEL therefore requires transformation of the EKF domain model into XSD.

The data types supported by the EKF domain language can directly be mapped to the data types as supported by the XSD (e.g. string, integer, double, float, datetime, etc.). The data types for which no direct mapping exist can be extended in the XSD type library. In the EKF domain language two distinct entities can contain attributes with the same name. The XSD enables this through the *namespaces*[1], as an unique identifier for an entity. An entity in the EKF domain models defines a complex structure (and associated behavior) that can directly be mapped by default onto the *complex type* in XSD. The attributes could optionally have a multiplicity indicated by [0..∗], which can directly be mapped to the *minOccurs* and *maxOccurs* in XSD. In Listing B.2 we give an example of the translation of the entity MortgageRequest into XSD.

```
<xs:ComplexType="MortageRequest">
 <xs:all>
  <xs:element name="debt_to_income_ratio" type="double"
   minOccurs="0" maxOccurs="1"/>
  <xs:element name="loan_to_value_ratio" type="double"
   minOccurs="0" maxOccurs="1"/>
  <xs:element name="loanamount" type="double"
   minOccurs="0" maxOccurs="1"/>
  <xs:element name="propertyvalue" type="double"
   minOccurs="0" maxOccurs="1"/>
  <xs:element name="monthlyincome" type="double"
   minOccurs="0" maxOccurs="1"/>
  <xs:element name="monthlydebt" type="double"
   minOccurs="0" maxOccurs="1"/>
        <xs:element name="insurancerate" type="double"
         minOccurs="0" maxOccurs="1"/>
        <xs:element name="taxrate" type="double"
         minOccurs="0" maxOccurs="1"/>
        <xs:element name="state" type="string"
         minOccurs="0" maxOccurs="1"/>
        <xs:element name="status" type="string"
         minOccurs="0" maxOccurs="1"/>
  <xs:element name="monthly_debt_list" type="double"
   minOccurs="0" maxOccurs="Unbound"/>
  <xs:element name="monthly_income_list" type="double"
   minOccurs="0" maxOccurs="Unbound"/>
 </xs:all>
</xs:ComplexType>
```

Listing B.2: Example translation of the entity and attributes into XSD

[1]see: XML Namespaces primer W3C at http://www.w3.org/TR/REC-xml-names/

In the EKF language the entities associate to other entities in the domain model. An association specifies the role of the relation and the multiplicity of the source and target entities. The association relation can be directly mapped onto a multiplicity relation of XSD complex type (see Listing B.3).

```
<xs:ComplexType="MortageRequest">
 <xs:all>
  <xs:element name="ResultsIn">
   <xs:complexType>
    <xs:sequence>
     <xs:element ref="Offer"
      minOccurs="1" maxOccurs="1"/>
    </xs:sequence>
   </xs:ComplexType>
  </xs:element>
  <xs:element name="BasedOn">
   <xs:complexType>
    <xs:sequence>
     <xs:element ref="Product"
      minOccurs="1" maxOccurs="1"/>
    </xs:sequence>
   </xs:ComplexType>
  </xs:element>
 </xs:all>
</xs:ComplexType>
```

Listing B.3: Example translation of the association relation into XSD

Generalization is a fundamental concept in the EKF domain model, for which the specialized entities, which inherit the attributes, operations and associations from its parent. For the generalization the complex type definition is abstracted using XSD property abstract and substitution group (see Listing B.4).

```

<!-- Generic product from case study -->
<xs:element="Product" type="Product" abstract="true"/>
<xs:ComplexType name="Product" abstract="true">
 <xs:sequence>
  <xs:element name="name" type="string"
   minOccurs="1" maxOccurs="1"/>
  <xs:element name="max_debt_to_income_ratio" type="double"
   minOccurs="1" maxOccurs="1"/>
  <xs:element name="max_loan_to_value_ratio" type="double"
   minOccurs="1" maxOccurs="1"/>
 </xs:sequence>
</xs:ComplexType>
<!--Product A from case study -->
<xs:element="ProductA" type="ProductA"
 subsitutionoGroup="Product"/>
<xs:ComplexType name="ProductA">
 <xs:sequence>
  <xs:complexContent>
   <xs:extension base="Product">
```

```
    <xs:sequence>
     <xs:element name="paymentform"
      minOccus="1" maxOccurs="1"/>
    </xs:sequence>
   </xs:extension>
   </xs:sequence>
</xs:ComplexType>
<!-- Product B from case study -->
<xs:element="ProductB" type="ProductA"
 subsitutionoGroup="Product"/>
<xs:ComplexType name="ProductB">
 <xs:sequence>
  <xs:complexContent>
   <xs:extension base="Product">
    <xs:sequence>
     <xs:element name="coverageform"
      minOccus="1" maxOccurs="1"/>
    </xs:sequence>
   </xs:extension>
  </xs:complexContent>
 </xs:sequence>
</xs:ComplexType>
<!--Product C from case study -->
<xs:element="ProductC" type="ProductC"
 subsitutionoGroup="Product"/>
<xs:ComplexType name="ProductC">
 <xs:sequence>
  <xs:complexContent>
   <xs:extension base="Product">
    <xs:sequence>
     <xs:element name="insuranceform"
      minOccus="1" maxOccurs="1"/>
    </xs:sequence>
   </xs:extension>
  </xs:complexContent>
 </xs:sequence>
</xs:ComplexType>
```

Listing B.4: Example translation of the generalization into XSD

In the EKF the enumeration refers to an element that specifies a bounded list of a certain data type. The XSD supports an enumeration of attributes in through the simple type (see Listing B.5).

```
<xs:simpleType name="propertyTypes">
 <xs:restriction base="xs:string">
  <xs:enumeration value="1 unit property"/>
  <xs:enumeration value="2 unit property"/>
  <xs:enumeration value="2 unit property"/>
  <xs:enumeration value="4 unit property"/>
 </xs:restriction>
</xs:simpleType>
```

Listing B.5: Example translation of the enumeration into XSD

The EKF domain model allows integrity rules to guard the consistency of the domain model through a set of predefined operations (e.g. Length, RegulairExpression and Between). The integrity rules of the EKF domain model can directly be mapped onto the restrictions in XSD (e.g. Length, maxLength, minLength, maxExclusive, minExclusive, Pattern, etc.) (see Listing B.6).

```
<!-- Example: Length(customer.name) < 20 -->
<xs:element name="name">
 <xs:simpleType>
  <xs:restriction base="xs:string">
   <xs:maxLength value="20"/>
   </xs:restriction>
  </xs:simpleType>
 </xs:element>
<!-- Example: ReguarExpression(Zipcode,Customer.zipcode)=TRUE -->
<xs:element name="zipcode">
 <xs:simpleType>
  <xs:restriction base="xs:string">
   <xs:pattern value="[0-9]{4}[A-Z]{2}"/>
  </xs:restriction>
 </xs:simpleType>
</xs:element>
<!-- Example: Between(Offer.riskrating,0,1000)=TRUE-->
<xs:element name="riskrating">
 <xs:simpleType>
  <xs:restriction base="xs:integer">
   <xs:minInclusive value="0"/>
   <xs:maxInclusive value="1000"/>
  </xs:restriction>
 </xs:simpleType>
</xs:element>
```

Listing B.6: Example translation of the integrity rules into XSD

The EKF process references to the domain data through a domain model instance. In BPEL the references to the case data is done through a correlation set, which specifies unique keys used to identify the case data (see Listing B.7).

```
<correlationSets>
 <correlationSet name="DataInstance" properties="DomainModelInstance"/>
</correlationSets>
```

Listing B.7: Mapping EKF data instance to correlation set in BPEL

In extend to observation above the attribute values of the EKF domain should be exposed as and XML following the definition of the XSD. In the BPEL definition the message ($x$) are assigned to BPEL variables (y) or the other way around (see Listing B.8). In extend to this approach, XSLT can be used to define transformations between one XSD into another XSD, allowing the transformation from one XML definition into another XML defintions. However, XSLT is out of the scope of this thesis.

```
<assing>
 <copy>
 <from>x</from><to>y</to>
 </copy>
</assign>
```

Listing B.8: Mapping EKF data instance to correlation set in BPEL

From these observations we can provide a meta-model presenting the mappings between the EKF domain and XSD (see Figure B.9).

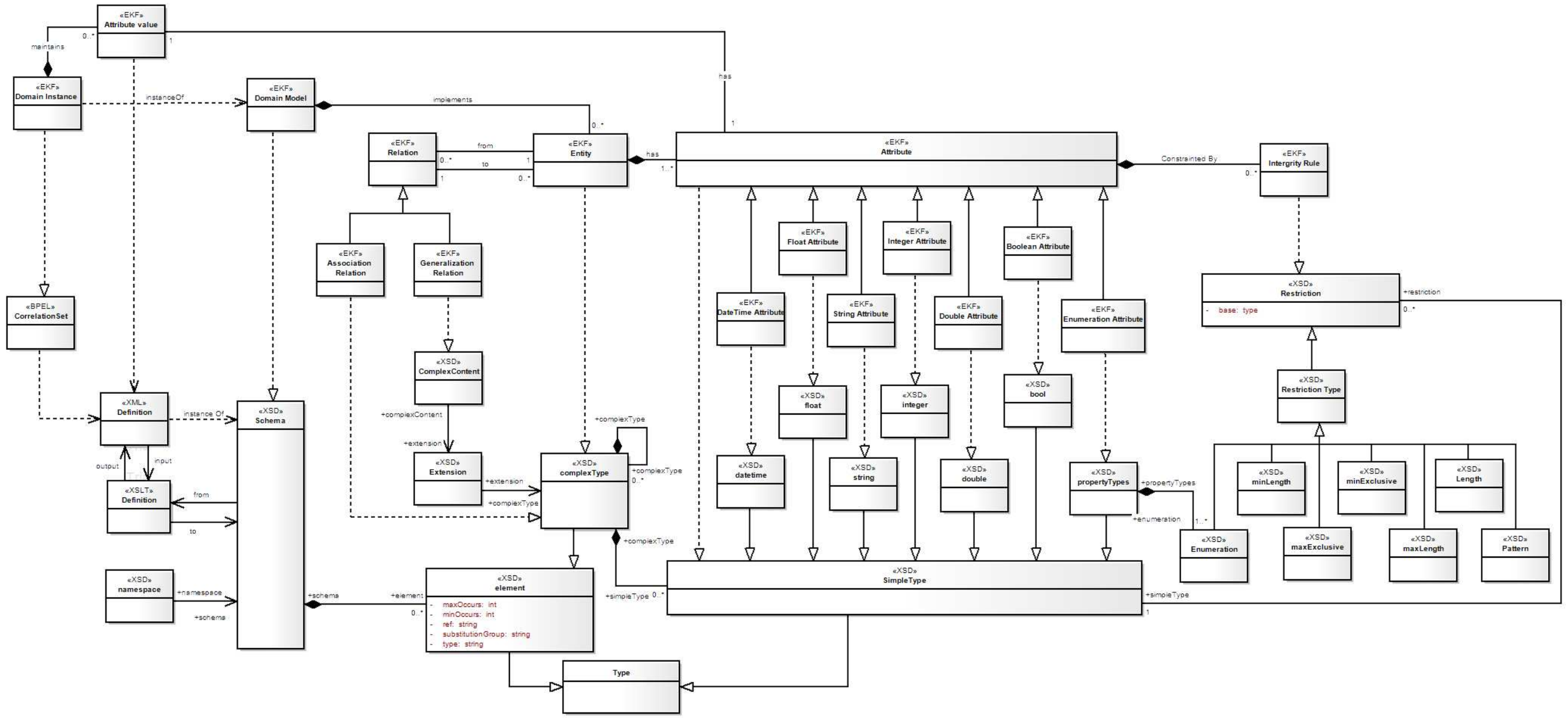


FIGURE B.9: Meta-model domain transformation of EKF and XSD.

# Appendix C

# Technical Spaces transformation

The aim of the technical spaces transformation is to determine for which process modeling concepts supported by the EKF, a solution can be found in terms of BPEL. We follow the classification of the service interaction and workflow patterns as intermediate concepts for the translation of the EKF process language concepts into BPEL. Enabling comparison of the semantics of the EKF process language we express the formal semantics of each concept in CPN, toward a first attempt to formalize the semantics of the EKF process language.

## C.1 Interaction Patterns

The interaction patterns specifies a number of concepts that describe the run-time message passing. Specifically, a set of service interaction patters supported by the EKF connectivity layer play an important role in the process. The following interaction patterns are considered: Request/Reply (CP1), One way (CP2), Synchronous Polling (CP3), Message Passing (CP4) and Publish/Subscribe (CP5).

**Request/Reply pattern (CP1)**

The request/reply is a form of synchronous communication, where a sender makes a request to a receiver and waits for a reply before continuing processing (see Figure C.1). The reply may influence further processing on the sender side.

**Solution of (CP1) in EKF**

In the EKF this communication pattern is modeled as presented in figure C.2, where an event $E1$ triggers the activation of a task $A$ and with a notification event $E2$ marks its completion.

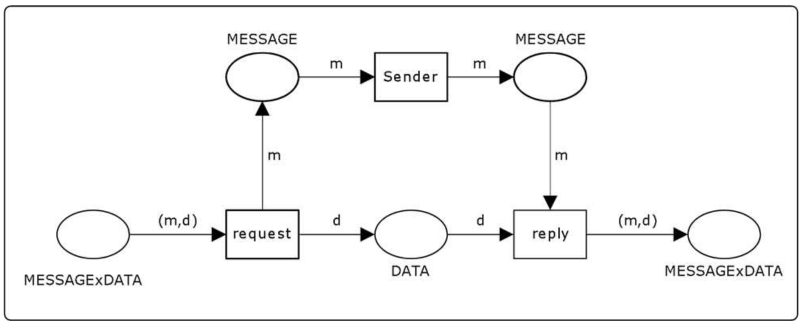


FIGURE C.1: CPN solution for CP1.

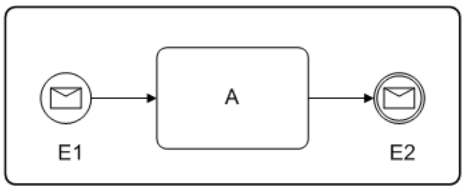


FIGURE C.2: EKF solution for CP1.

### Solution of (CP1) in BPEL

In BPEL the request/reply is modeled by the invoke activity (see Listing C.1). Furthermore, the invoke activity must specify two different variable: one input-variable, where the outgoing data from the process is stored (or input data for the communication); and one output-variable, where the incoming data is stored (or the output data for the communication).

```
<sequence>
 <invoke operation="A" inputvariable="Request" outputvariable="Result"/>
</sequence>
```

LISTING C.1: BPEL code for CP1

### One way pattern (CP2)

The one way pattern is a form of synchronous communication where a sender makes a request to a receiver and waits for a reply that acknowledges the receipt of the request. Since the receiver only acknowledge the receipt, the reply only delays further processing on the sender side.

### Solution of (CP2) in EKF

In the EKF this pattern can be modeled as presented in figure C.4, where the receiver waits for the reception of an event $E1$, which afterward directly result in sending a notification message $E2$.

### Solution of (CP2) in BPEL

The one way pattern differs from the request/reply implementation in BPEL (see Listing

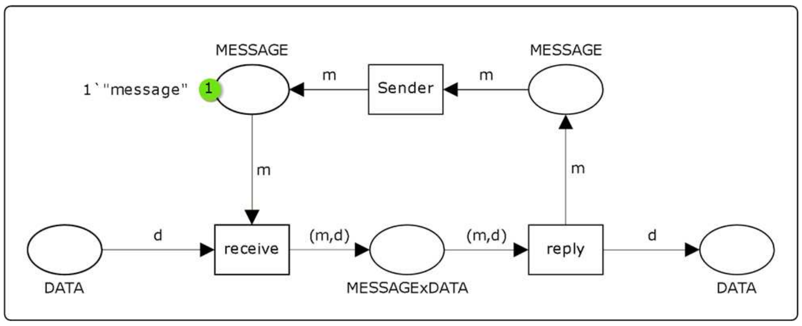


Figure C.3: CPN solution for CP2.

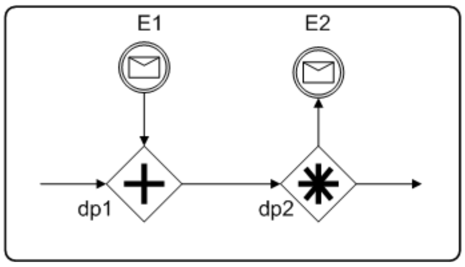


Figure C.4: EKF solution for CP2.

C.2) by sending a reply immediately after the reception of the message (i.e. no processing is performed between the receipt and reply activities).

```
<sequence>
 <receive operation="E1"/>
 <reply operation="E2"/>
</sequence>
</scope>
```

Listing C.2: BPEL code for CP2

### Synchronous polling pattern (CP3)

This communication pattern is a form of synchronous communication, where a sender communicates a request to a receiver, but instead of blocking, continues processing. At intervals the sender checks to see if a reply has been send. When it detects a reply it directly stops any further polling for a reply.

### Solution of (CP3) in EKF

This pattern can not be modeled with the EKF process language. The problem is caused by the lack of support for a synchronization point in the EKF process language, which is required to define at what point the process polling result should be finally accepted.

### Solution of (CP3) in BPEL

In BPEL this pattern is captured through the utilization of two parallel flows: one for the receipt of the expected response, and one for the sequence of activities not depending

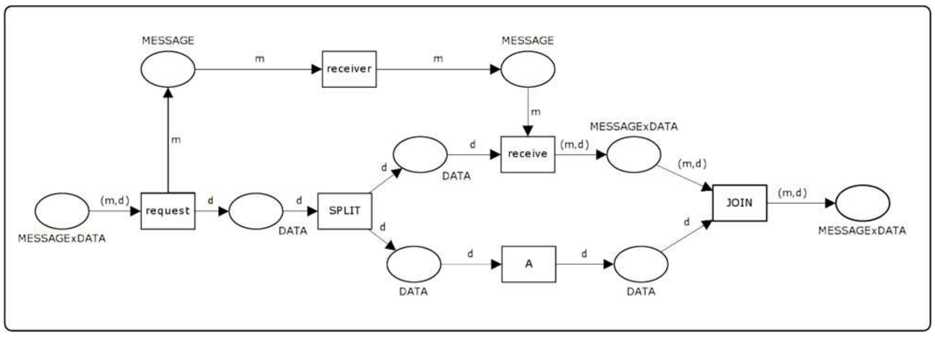


Figure C.5: CPN solution for CP3.

on the this response (see Listing C.3). The initialization of the communication patterns is done beforehand through an invoke action. To be able to proceed, the invoke action is specified to send data and not wait for a reply. This is indicated by the use of an input-variable and by omitting the specification of an output-variable.

```
<sequence>
 <invoke operation="E1"  ... inputvariable="Request"/>
 <flow>
  <sequence> ... </sequence>
  <receive operation="E1" ... outputvariable="Result"/>
 </flow>
<sequence>
```

Listing C.3: BPEL code for CP3

### Message passing pattern (CP4)

Message passing is a form of a-synchronous communication where a request is send from a sender to a receiver. When the sender has made the request, it essentially forgets it has been send and continues processing (see Figure C.6).

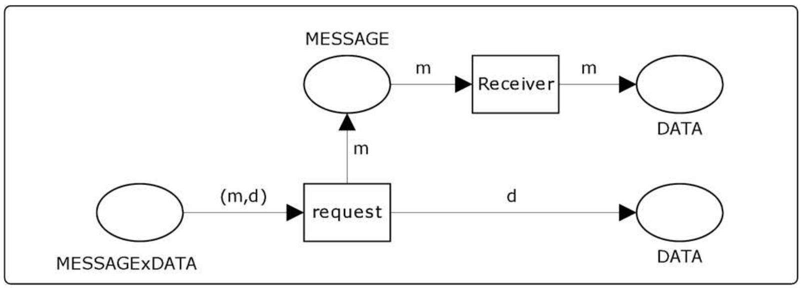


Figure C.6: CPN solution for CP4.

### Solution of (CP4) pattern in EKF

The message passing is directly supported by the process language EKF through the notification message as presented in figure C.7. The notification message triggers an external system or process, without any requirement of synchronization.

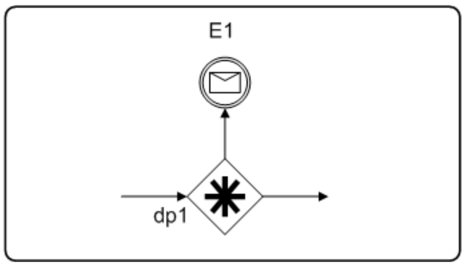


FIGURE C.7: EKF solution for CP4.

**Solution of (CP4) in BPEL**

This solution is has already been demonstrated as part of the solution of CP3, namely the invoke activity with only an input-variable (see Listing C.3).

## C.2 Basic control flow patterns

This class of patterns captures the elementary aspects of the process control. The basic control flow patterns include the sequence pattern (WP1), parallel split (WP2), synchronization merge (WP3), exclusive choice (WP4) and simple merge (WP5).

**Sequence pattern (WP1)**

The sequence pattern is defined as being an ordered series of tasks, for which one task is started after the completion of a previous task. The semantics of the messages and timers in a sequence result in respectively waiting for the reception of the message or the expiration of a timer.

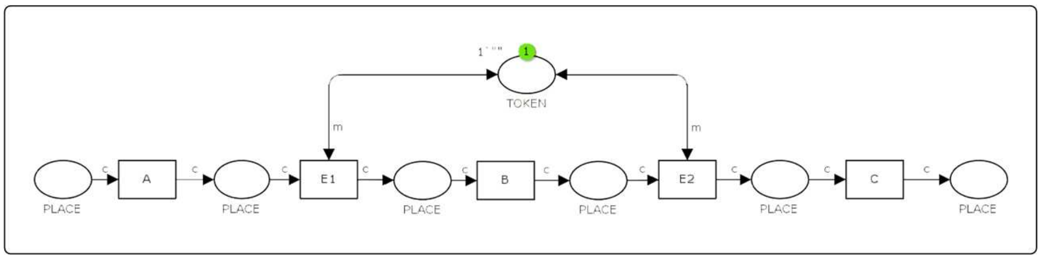


FIGURE C.8: CPN solution for WP1.

**Solution of (WP1) in EKF**

In the EKF process language a sequence is modeled as a series of preceding tasks or intermediate events with one incoming and one outgoing (unconditional) flow (see Figure C.9). The flow constraints the order in which the task need to be executed, or events must be processed.

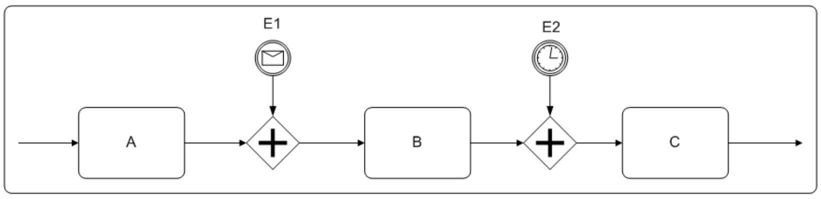


FIGURE C.9: EKF solution for WP1.

### Solution of WP1 in BPEL

In BPEL a sequence pattern is implemented as a sequence activity containing one or more invoke, receive and wait activities (see Listing C.4).

```
<sequence>
 <invoke operation="A"/>
 <receive operation="E1"/>
 <invoke operation="B"/>
 <wait name="E2"/>
 <invoke operation="C"/>
</sequence>
```

LISTING C.4: BPEL solution for WP1

### Parallel split (WP2)

The parallel split pattern is defined as the divergence of a branch into two or more parallel branches each are performed concurrently.

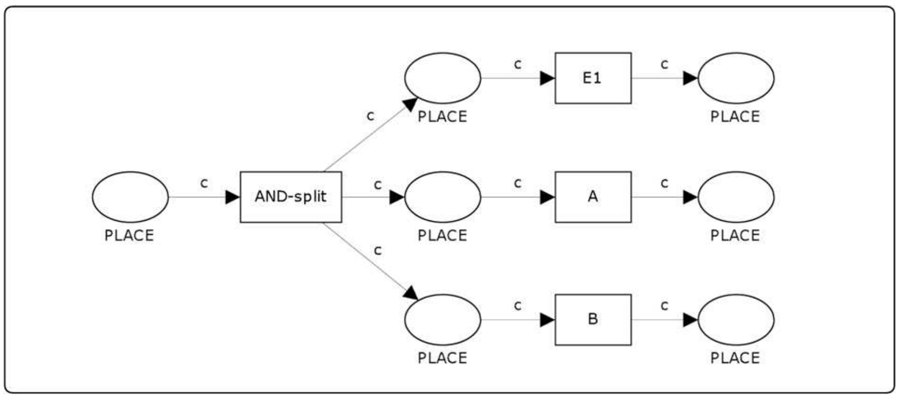


FIGURE C.10: CPN solution for WP2.

### Solution of WP2 in EKF

In the EKF process language this can be modeled as an OR-split-decision-point with multiple unconditional outgoing flows (see Figure C.11). The unconditional flow specifies that all outgoing flows are activated. The OR-split-decision-point allows multiple tasks, notification messages and occurrence of intermediate events, to be controlled in parallel. Notice that notification messages does not contain a post-condition, such that after the completion of the interaction, the branches with a notification message are considered to be completed.

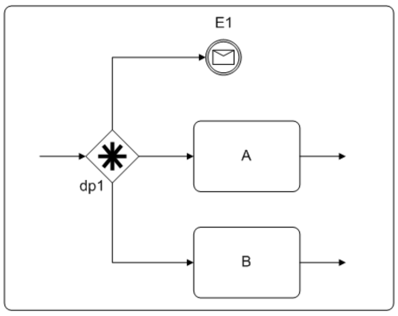


Figure C.11: EKF solution for WP2.

**Solution of WP2 in EKF**

The parallel split is directly supported by the BPEL flow activity, for which all activities contained by the flow are executed in parallel (see Listing C.5). Particularly, the parallel split does not require additional join-conditions or links.

```
<flow>
 <invoke operation="E1"/>
 <invoke operation="A"/>
 <invoke operation="B"/>
</flow>
```

Listing C.5: BPEL code for WP2

**Synchronization merge (WP3)**

The synchronization merge pattern is defined as the converge of two or more branches into a single subsequent branch, such that the thread of control is passed to the subsequent branch when all input branches have been enabled. The synchronization merge is the result of a parallel split earlier in the process and waits for the completion of all the branches in order to proceed processing (see Figure C.12).

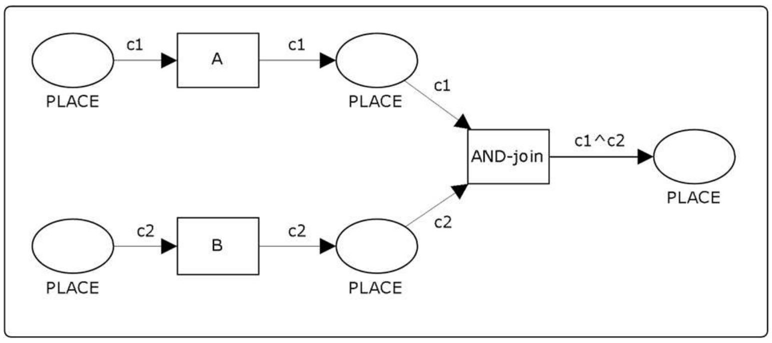


Figure C.12: CPN solution for WP3.

**Solution of WP3 in EKF**

This pattern is not supported by the EKF process engine, as the EKF process engine does not allow the synchronization of multiple parallel branches. This is mainly caused by an implementation limitation of the EKF process engine.

### Solution of WP3 in BPEL

In BPEL the synchronization merge is directly supported though the flow activity, for which each contained task waits for the completion of all other tasks nested in the flow as presented in listing C.6.

```
<flow>
 <invoke operation="A"/>
 <invoke operation="B"/>
</flow>
```

Listing C.6: BPEL code for WP3

### Exclusive choice pattern (WP4)

The exclusive choice pattern is defined as being a location in the process where the flow is split into two or more exclusive alternative paths. The branches are exclusive such that only one of the alternative paths may be chosen to continue the process (see Figure C.13).

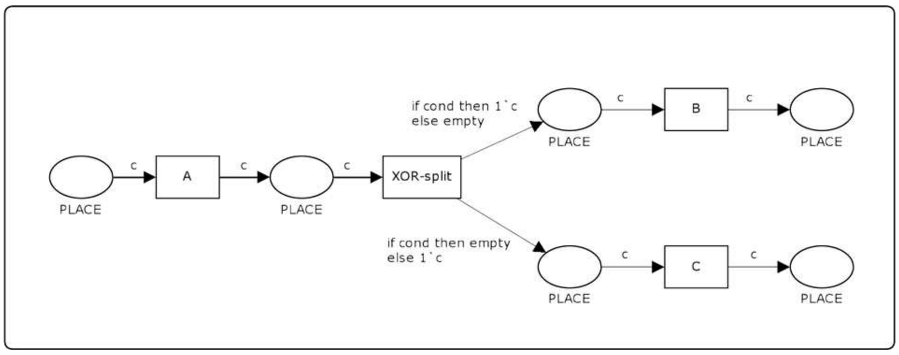


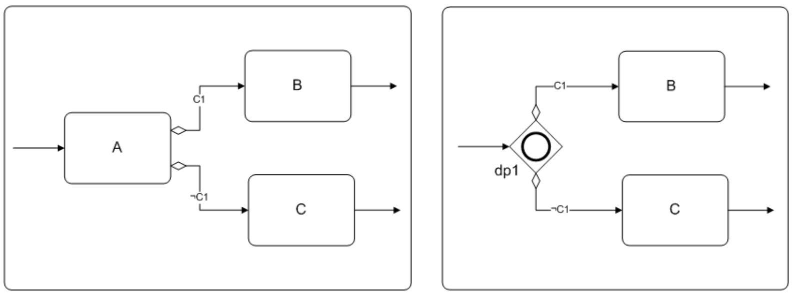
Figure C.13: CPN solution for WP4.

### Solution of WP4 in EKF

Since additional control is needed to create an exclusive choice pattern, the EKF process language uses multiple conditional outgoing flows from either a task or a conditional XOR-split-decision-point. The conditional flow is governed by a condition expression, which will be preceded only if its evaluates to true. This implies that after the completion of each task the expressions are evaluated and only the first flow for which the condition expression evaluates to true is preceded, such that all other paths are discarded. Two approaches exist in the EKF process language to model this pattern: in the first alternative the exclusive choice is performed directly after the completion of a task (see Figure C.14(a)) and the second alternative the exclusive choice is explicitly modeled an an XOR-split-decision-point (see Figure C.14(b)).

### Solution of WP4 in BPEL

In BPEL the exclusive choice is modeled as a switch activity, where a condition defines the preceding of the process based on the evaluation of exclusive conditions. The first



(a) Exclusive Choice (alternative a) (b) Exclusive Choice (alternative b)

Figure C.14: EKF solution for WP4.

case element in the switch activity from which the condition evaluates to true will be considered as the preceding of the process. An otherwise element is required in the switch activity, specifies the alternative path, which must be available in case case none of the case conditions are satisfied. In BPEL the otherwise is requires to avoid deadlocks. The BPEL translation for the alternative (a and b) is presented in listing C.7. Notice that the main difference between both approaches (a) and (b) is that (a) requires that a task is performed before exclusive choice. Line 1,2 and 13 in Listing C.7 are therefore only required for alternative (a).

```
 1 <sequence>
 2  <invoke operation="A"/>
 3  <switch>
 4   <case condition="C1">
 5    <invoke operation="B"/>
 6    ...
 7   </case>
 8   <otherwise>
 9    <invoke operation="C"/>
10    ...
11   </otherwise>
12  </switch>
13 </sequence>
```

Listing C.7: BPEL code for WP4

### Simple Merge pattern (WP5)

The simple merge pattern is defined as being a location in the process where a set of exclusive paths are joined into a single path (see Figure C.15).

### Solution of WP5 in EKF

The EKF process language supports two approaches for modeling the simple merge pattern (see Figure C.16(a) and C.16(b)). In alternative (a) two or more exclusive sequence flows are merged directly on a task, resulting in the immediate start of a task in case one of the branches become active. Alternative (b) allows the merge of two or

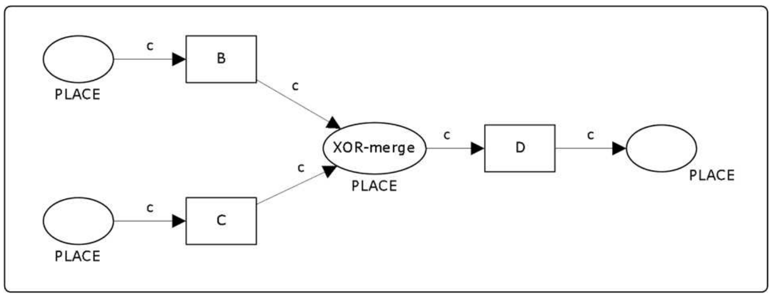
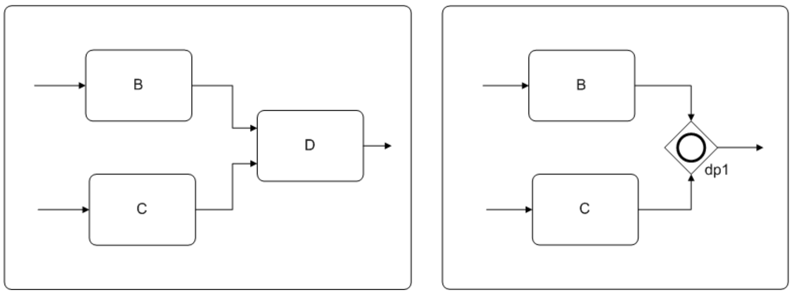


FIGURE C.15: CPN solution for WP5.

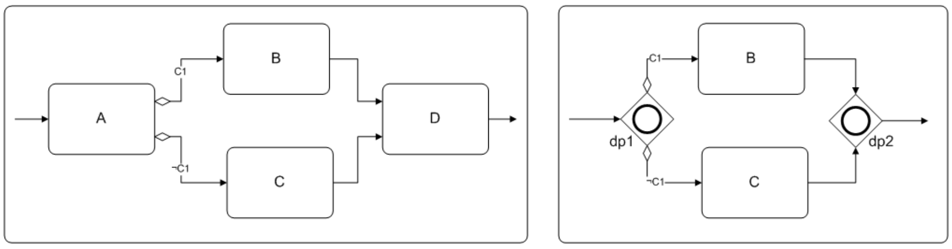
more branches one an XOR-merge-decision-point, for which the outgoing flow specifies a single point from which the process continues.



(a) Simple Merge (alternative a) (b) Simple Merge (alternative b)

FIGURE C.16: EKF solution for WP5.

A structured exclusive choice is composed of a single exclusive choice for which all branches are merged into a single simple merge, such that the process concept is composed of exactly one entry point and one exist point (see Figure C.17(a) and C.17(b)).



(a) Structured Exclusive Choice (Alternative a) (b) Structured Exclusive Choice (Alternative b)

FIGURE C.17: EKF solution for WP5.

### Solution of WP5 in BPEL

The structured exclusive choice can directly be mapped onto the BPEL switch activity (see Listing C.8). Notice that the lines 1, 2, 11 and 12 are only required when translating the alternative (a). The more complex combination of exclusive choice(s) and simple merge(s) are defined to be part of a more complex process structure namely the arbitrary cycle (WP10) pattern.

```
<sequence>
 <invoke operation="A"/>
 <switch>
  <case condition="C1">
   <invoke operation="B"/>
  </case>
  <otherwise>
   <invoke operation="C"/>
  </otherwise>
 </switch>
 <invoke operation="D"/>
</sequence>
```

Listing C.8: BPEL code for WP5

## C.3 Advanced branching and synchronization patterns

The advanced branching and synchronization patterns present the characteristics of the more complex branching and merging concepts, supported by the EKF process. The following patterns are considered: Multi-choice (WP6), Synchronizing Merge (WP7), Multi-merge (WP8) and Structured Discriminator (WP9).

### Multi-choice pattern (WP6)

The intent of the multi-choice pattern is to choose one or more branches, in which each branch is taken only if it satisfies a particular condition. A multi-choice is a composition of parallel-split and exclusive-choice patterns, such that one or more branches are enabled after the evaluation of a condition (see Figure C.18). The multi-choice only specifies how split of the branches is accomplished, not how they are eventually joined.

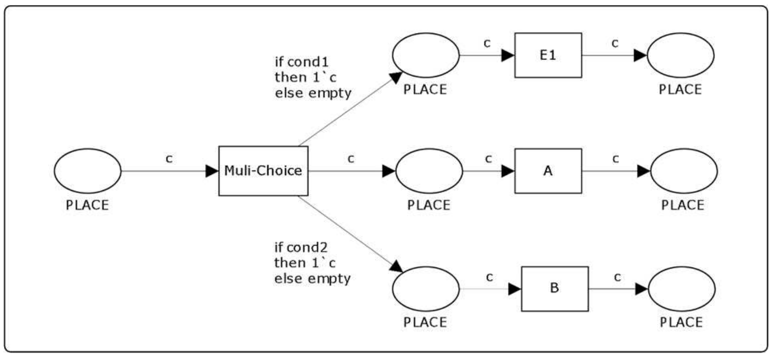


Figure C.18: CPN solution for WP6.

### Solution of WP6 in EKF

In the EKF process language the multi-choice is modeled as an OR-split-decision-point, which allows the activation of multiple conditional or unconditional flows (see Figure

C.19). Notice that the conditional flows does not satisfy the property to be exclusive, such that a deadlock can occur when none of the conditions on the branches are satisfied. In the EKF the conditions must satisfy the property that minimal one branch is preceded, either directly through an unconditional flow or indirectly through the composition of the conditions.

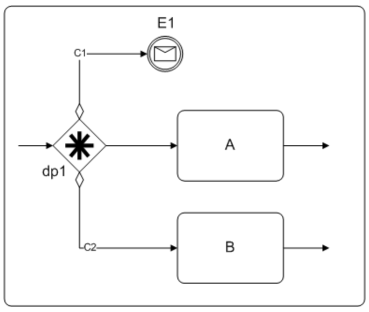


FIGURE C.19: EKF solution for WP6.

### Solution of WP6 in BPEL

In BPEL the multi-choice can be implemented as a flow consisting of join-conditions (see Listing C.9). The flow activity waits for each contained activity to complete (i.e. joins them) before exciting. The BPEL flow mechanism offers several features to model the situation where one activity cannot start until one or more activities on which they depend are completed. Each conditional branch of the multi-choice is implemented as a join-condition, guarding the activation of that specific branch. Notice that unconditional flow of the EKF process language does not require such a join-condition, because they are directly activated. The flow activity requires the property *suppressJoinFailure* set to *yes* support the *death path elimination semantics*[1] preventing deadlocks.

```
<flow supressJoinFailure="yes">
 <invoke operation="E1" joinCondition="C1"/>
 <invoke operation="A"/>
 <invoke operation="B" joinCondition="C2"/>
<flow>
```

LISTING C.9: BPEL code for WP6

### Synchronizing merge (WP7)

The structured synchronizing merges specifies that two or more branches (which diverged earlier in the process at a uniquely identifiable point) into a single subsequent branch such that the incoming tread of control is passed to the subsequent branch when each active incoming branch has been enabled (see Figure C.20).

[1]Defines that when a transition condition evaluates to false, the link to the target is discarded.

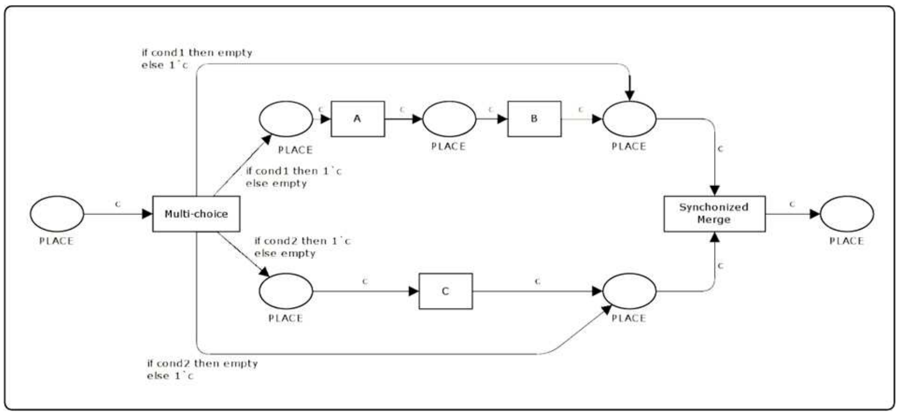


FIGURE C.20: CPN solution for WP7.

### Solution of WP7 in EKF

The EKF process language does not allow the synchronization of multiple parallel branches. This pattern can therefore not be modeled with the EKF process language.

### Solution of WP7 in BPEL

In BPEL this pattern is directly supported by the flow activity (see Listing C.10). The semantics of the flow requires that all including activities able to execute need to be fulfilled in order to complete the flow. Understanding flow in BPEL requires insight into the order in which the activities are executed, and whether a particular branch is even performed at al. Specifically, a flow can defines a set of links, each originating from a source activity in the flow and termination at a target activity in the flow. The target links must be satisfied before the subsequent activity can be activated. The links can even be guarded by a transition-condition (optional), such that the transition-condition must be satisfied in order to activate the link between the activities.

```
<flow suppressJoinFailure="yes">
 <links>
  <link name="AB">
 </links>
 <invoke operation="A" joinCondition="C1">
  <source linkName="AB" transitionCondition="...">
 </invoke>
 <invoke operation="B" joinCondition="C2"/>
  <target linkName="AB"/>
 </invoke>
 <invoke operation="C" joinCondition="C2"/>
</flow>
```

LISTING C.10: BPEL code for WP7

### Multi-merge (WP8)

The multi-merge specifies the converge of two or more branches into a single subsequent branch such that each enabling branch results in an instance being passed to the subsequent branch.

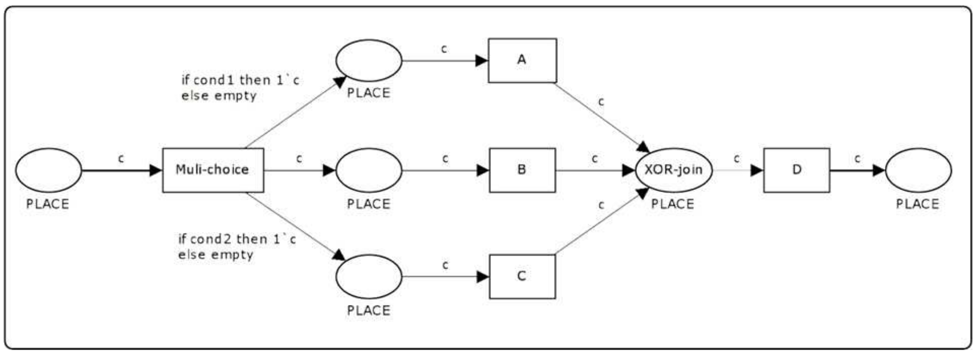


FIGURE C.21: CPN solution for WP8.

### Solution of WP8 in EKF

This pattern is not supported by the EKF process, although modeling this pattern in the EKF looks quite straight forward (e.g. merging multiple parallel branches through an XOR-merge-decision-point). The lack of support of this pattern is mainly caused by the limitation of the EKF process engine, as it does not allow the creation of multiple instances of tasks in the sub-process.

### Solution of WP8 in BPEL

This pattern is not directly supported by the structured activities in BPEL, because it is not allow two activate threads following the same path without creating new instances of another process. Alternatively, this pattern can be implemented with a flow for which multiple instances are created (see Listing C.11) through an external process (see Listing C.12). Some complexity issues arise as this solution requires an external process to accomplish the sematnics of WP8.

```
<flow suppressJoinFailure="yes">
 <links>
  <link name="AX">
  <link name="BX">
  <link name="CX">
 </links>
 <invoke operation="A" joinCondition="C1">
  <source linkName="AX">
 </invoke>
 <invoke operation="B">
  <source linkName="BX">
 </invoke>
 <invoke operation="C" joinCondition="C2">
  <source linkName="CX">
 </invoke>
 <invoke operation="Start(X)">
  <target linkName="AX"/>
 </invoke>
 <invoke operation="Start(X)">
  <target linkName="BX"/>
 </invoke>
 <invoke operation="Start(X)">
  <target linkName="CX"/>
 </invoke>
</flow>
```

Listing C.11: BPEL code for WP8 (part I)

```
<process>
  <receive operation="Start(X)" createInstance="yes">
  <invoke operation="D"/>
  ...
</process>
```

Listing C.12: BPEL code for WP8 (part II)

### Structured Discriminator (WP9)

The structured discriminator specifies the converge of two or more branches into a single subsequence branch following a corresponding divergence earlier in the process model, such that the thread of control is passed to the subsequent branches when the first incoming branch has been enabled. Subsequent elements of incoming branches do not result in the thread of control being passed, but they continue processing until they are completed (see Figure C.22).

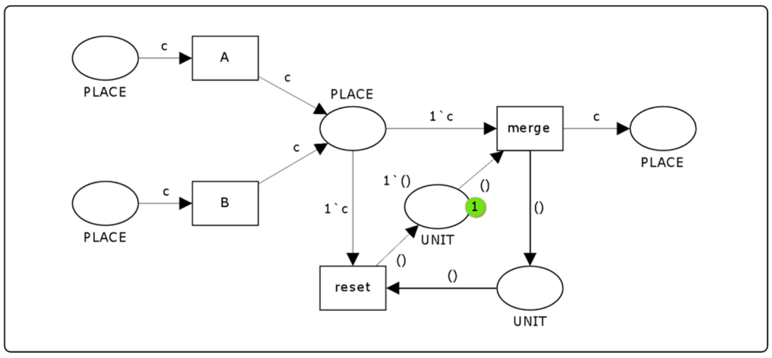


Figure C.22: CPN solution for WP9.

### Solution of WP9 in EKF

This pattern is modeled as an OR-split-decision point which activates the execution of multiple parallel tasks (see Figure C.23). The first task which completes triggers the continuation of the process, while the parallel running tasks continue processing until they are completed. This behavior is accomplished through a condition, which is assigned to false after the completion of the first task. Notice that this solution differs from the formal semantics of WP9 as this pattern is only supported for tasks contained by sub-processes, caused by the limitation of the EKF process engine.

### Solution of WP9 in BPEL

This pattern is not supported by a flow activity construct in BPEL. The reason for not

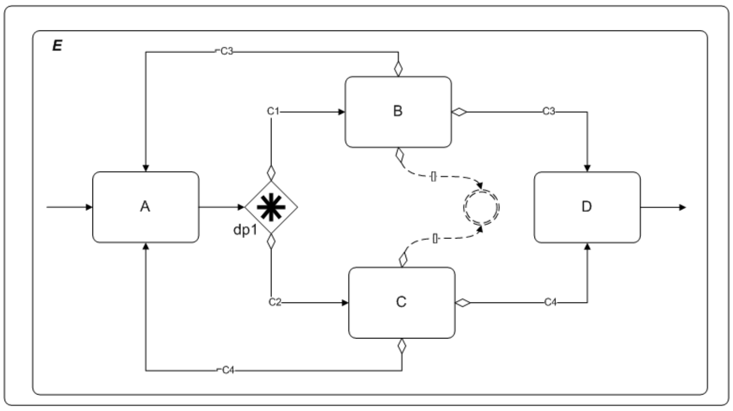


FIGURE C.23: EKF solution for WP9.

being able to use the link construct in combination with a join-condition, is caused by the fact that a join-condition is evaluated first, when the status of all incoming links are determines and not, as required in this case, when the first positive link is determined.

## C.4 Iteration patterns

The interaction patterns deal with the repetitive behavior in the process, the following iteration patterns are considered: Arbitrary cycles (WP10) and Structured loops (WP21).

**Arbitrary cycle (WP10)**
The arbitrary cycle pattern is a mechanism for allowing a certain segment of the process to be repeated. The arbitrary cycle pattern repeats a task or set of tasks by cycling back to it in the process (see Figure C.24).

**Solution of WP10 in EKF**
Figure C.25 gives an example of an arbitrary pattern modeled in the EKF process language (see Figure C.25). Notice that the arbitrary cycle is composed of a combination of: exclusive choice, deferred choice and simple merge patterns. The arbitrary cycle can be identified on its property to allow revisiting certain points in the process multiple times.

**Solution of WP10 in BPEL**
The arbitrary cycle is not directly supported by the basic activities in BPEL. Although the while activity allows structured loops (WP21), it is not possible to jump back to arbitrary parts in the process (i.e. only loops with one entry point and one exit point are allowed). Alternatively, one can implement the arbitrary cycle to take advantage of the BPEL event handler. The arbitrary cycle is translated into a scope activity which is

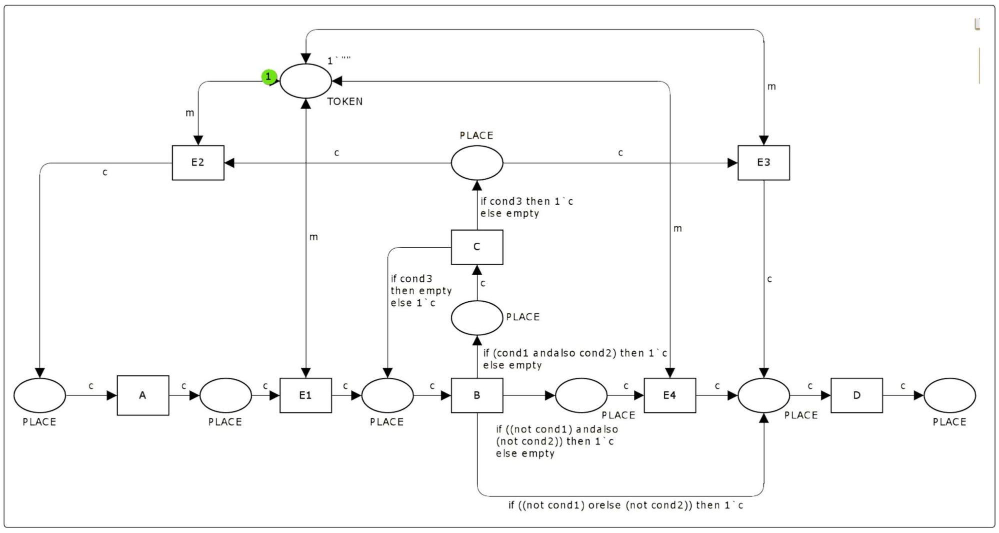


FIGURE C.24: CPN solution for WP10.

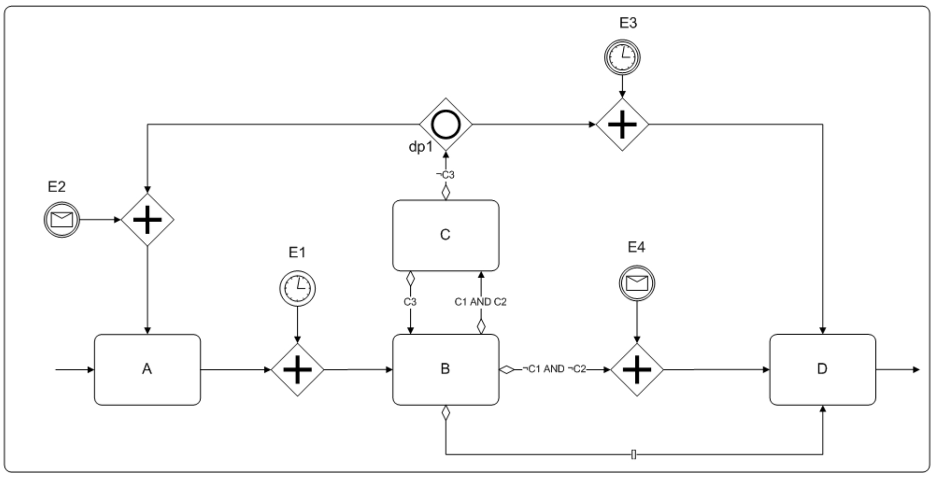


FIGURE C.25: EKF solution for WP10.

triggered by an event handler (Start($A$) in our example) and continues waiting until it receives an end event which is the event (End($D$) in our example). This implementation allows that each point in the process which resembles to the event handler can be revisited. In this way the event handler is used as a return point which is called through the invoke activity (see Listing C.13). This solution results in several complexity issues as it is based on the abuse of the event handler. These complexity issues directly arise from the technical limitations of the event handler in BPEL. The event handler is exposed as a Web service as part of the process to which it belongs. BPEL does not allow that Web services that belong to a process are invoked from the process itself. This limitation requires that an external process is required to process to create stubs that control the event handlers of the arbitrary cycles.

```
<scope name="WP10">
 <eventHandler>
```

```
  <!-- Revisit of activity A -->
  <OnMessage operation="Start(A)">
         <sequence>
    <invoke operation="A"/>
    <wait name="E1"/>
    <invoke operation="Start(B)"/>
   </sequence>
  </OnMessage>
  <!-- Revisit of activity B -->
  <OnMessage operation="Start(B)"/>
   <sequence>
    <invoke operation="B"/>
     <switch>
      <case condition="C1 and C2">
       <invoke operation="Start(C)"/>
      </case>
      <case condition="C1 and C2">
       <sequence>
        <receive operation="E4"/>
        <invoke operation="Start(D)"/>
       </sequence>
      </case>
      <otherwise>
       <invoke operation="Start(D)"/>
      </otherwise>
     </switch>
    </sequence>
   </OnMessage>
   <!-- Revisit of activity C -->
   <OnMessage operation="Start(C)">
    <sequence>
     <invoke operation="C"/>
     <switch>
      <case condition="C3">
       <invoke operation="Start(B)"/>
      </case>
      <otherwise>
       <pick>
        <onMessage operation="E2">
         <invoke operation="Start(A)"/>
        </onMessage>
        <onAlarm name="E3">
         <invoke operation="Start(D)"/>
        </onAlarm>
       </pick>
      </otherwise>
     </switch>
    </sequence>
   </OnMessage>
   <!-- Revisit of activity A -->
   <OnMessage operation="Start(D)">
    <sequence>
     <invoke operation="D"/>
     <invoke operation="End(D)"/>
    </sequence>
   </OnMessage>
  </eventHandler>
  <!-- Start point for the scope -->
  <sequence>
   <invoke operation="Start(A)"/>
   <receive operation="End(D)"/>
  </sequence>
</scope>
```

Listing C.13: BPEL code for WP10

### Structured Loops (WP21)

The structured loops allows performing tasks or sub-processes repeatedly until either an associated pre-condition (at the beginning of of the loop) or post-condition (at the end of the loop) is satisfied. Three types of structured loops can be identified: repeat until, while and while + repeat loop. The *repeat until loop* requires a task to be performed minimal once and is repeated until a certain condition has been satisfied (see Figure C.26(a)). This is similar to the *while loop*, but different from the repeat as the task must be performed minimal once (see Figure C.26(b)). The *while + repeat loop* is a combination of both the repeat until and while loop, composed of two tasks. Take for example the tasks A and B of figure C.26(c). Task A is performed minimal once, followed by performing a sequence of the task A and B until a certain post-condition is satisfied.

### Solution of WP21 in EKF

The three structured loop types are directly supported by the EKF process language (see Figure C.27(a), C.27(b) and C.27(c)).

### Solution of WP21 in BPEL

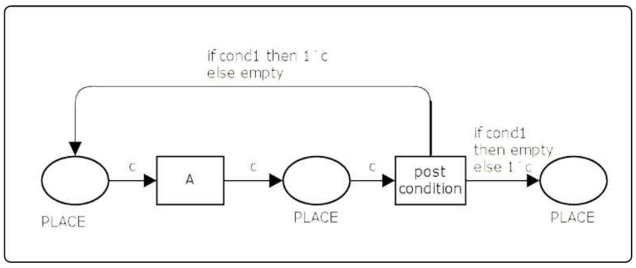
The various structured loops are also directly supported by the basic activities of BPEL (see Listing C.14, C.15 and C.16). Notice that the repeat until and repeat + while loop require multiple invocations of the same task.

```
<sequence>
 <invoke operation="A"/>
 <while condition="C1">
  <invoke operation="A"/>
 </while>
</sequence>
```

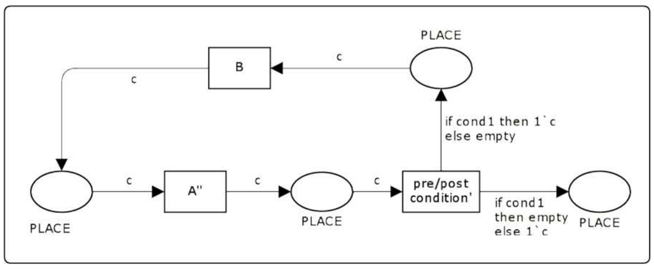
Listing C.14: BPEL code for WP21(c) (Repeat Until)

```
<while condition="C1">
 <invoke operation="A"/>
</while>
```

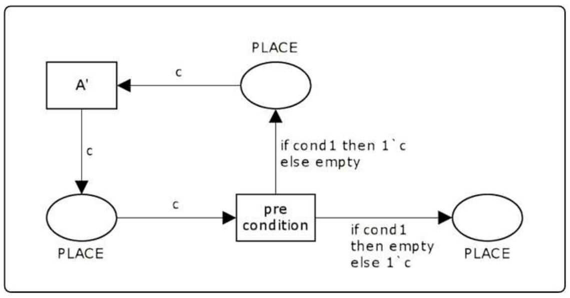
Listing C.15: BPEL code for WP21(b) (While)



(a) Repeat Until

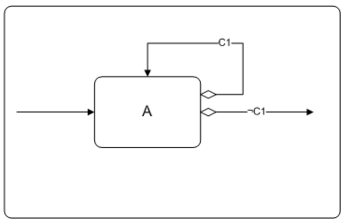


(b) While

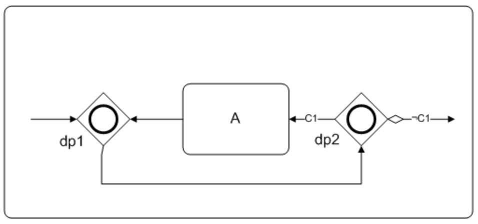


(c) Repeat + While

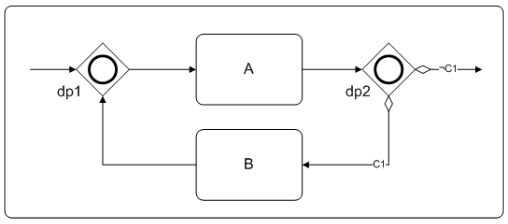
Figure C.26: CPN solutions for WP21.



(a) Repeat Until



(b) While



(c) Repeat + While

Figure C.27: EKF solutions for WP21.

```
<sequence>
 <invoke operation="A"/>
 <while condition="C1">
  <sequence>
   <invoke operation="A"/>
   <invoke operation="B"/>
  </sequence>
 </while>
</sequence>
```

Listing C.16: BPEL code for WP21(c) (Repeat + While)

## C.5 Multiple Instance Patterns

The multiple instance patterns describe the situations where there are multiple threads of control active in the process model, which relate to the same task (hence share the same implementation definition). We consider the following multiple instance patterns: Multiple Instances without Synchronization (WP12), Multiple Instances with Priori Design-Time Knowledge (WP13) and Multiple Instances with a Priori Run-Time Knowledge (WP14).

### MI without synchronization (WP12)

The intent of the multiple instance without synchronization pattern to perform multiple concurrent instances of a task, but let each run on its own with no overall synchronization (see Figure C.28). The instance might be created consecutively, but they will be able to run in parallel, which distinguishes this pattern from the arbitrary cycle pattern (WP10).

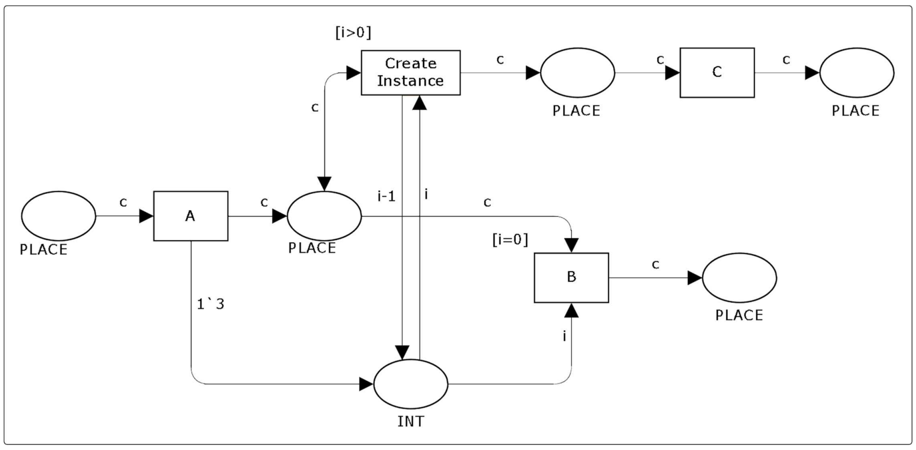
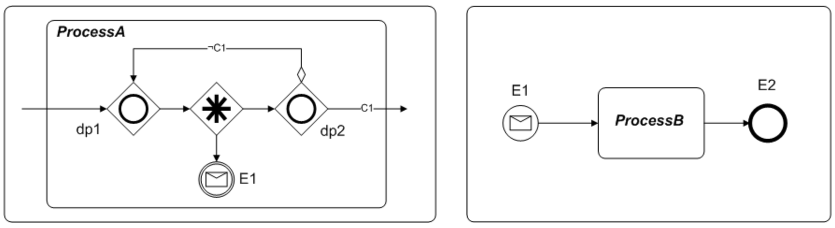


Figure C.28: CPN solution for WP12.

### Solution of WP12 in EKF

With the EKF process language this pattern can be modeled as two processes for which the first process (see Figure C.29(a)), contains a while loop, which creates multiple instance of another process (see Figure C.29(b)).



(a) Part I (b) Part II

FIGURE C.29: EKF solutions for WP12.

### Solution of WP12 in BPEL

In BPEL this pattern can be created by using the invoke activity embedded in a while loop (see listing C.17 and C.18). The creation of a new process instance is accomplished at line 4 of listing C.17, triggering the process of listing C.18.

```
1 <sequence>
2  <invoke operation="A"/>
3  <while condition="C1"/>
4   <invoke operation="Start(X)"/>
5  </while>
6  <invoke operation="B"/>
7 </sequence>
```

LISTING C.17: BPEL code for WP12 (Part I)

```
1 <process>
2  <receive operation="Start(X)"createInstance="yes">
3  <invoke operation="B">
4  ...
5 </process>
```

LISTING C.18: BPEL code for WP12 (Part II)

### MI with a Priori Design-Time Knowledge (WP13)

The intent of the multiple instances with a priori design-time knowledge pattern is to perform $i$ multiple concurrent instances of a task, where $i$ is a constant at run-time. In addition, it joins these instances before continuing with a remainder of the process (see Figure C.30).

### Solution of WP13 in EKF

This pattern is not supported by the EKF process language, as it does not allow multiple instances of a task in the process. Alternatively, the number of task could be replicated as many times as needed, but this is not allowed as the EKF process language does not allow the synchronization of parallel branches.

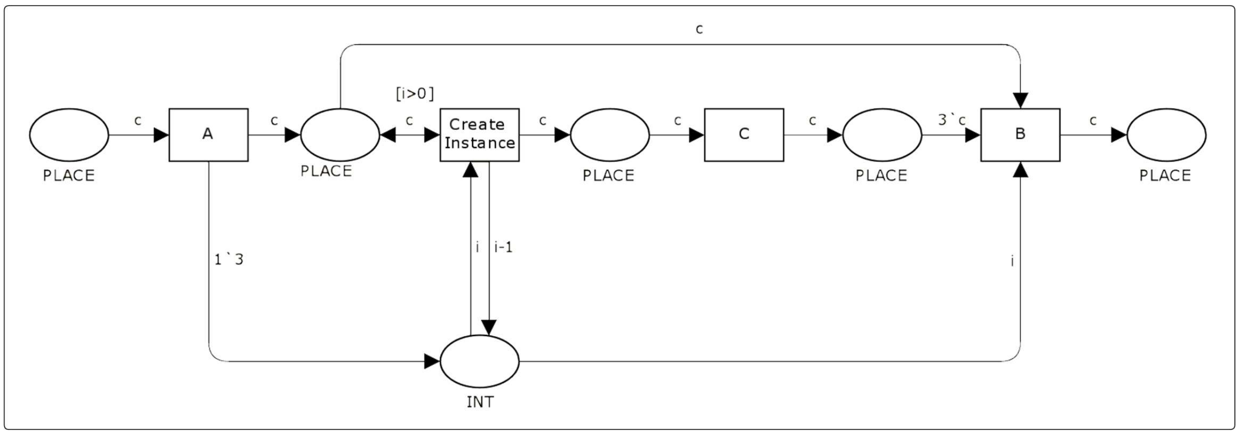


FIGURE C.30: CPN solution for WP13.

## Solution of WP13 in BPEL

If the number of instances to be synchronized at design-time, a simple solution is to replicate the activity as many times as it needs to be instantiated, and run the replicas in parallel by placing them in a flow activity (see Listing C.19). Different implementations of BPEL have introduced language extensions to directly support this pattern (e.g. flowN activity of Oracle BPEL).

```
<!-- Create 3 instances -->
<flow>
 <invoke operation="C">
 <invoke operation="C">
 <invoke operation="C">
</flow>
```

LISTING C.19: BPEL code for WP13

## MI with a Priori Run-Time Knowledge (WP14)

The intent of the multiple instances with a priori run-time knowledge is to perform $i$ multiple concurrent instances of a task, where the value of $i$ is known at run-time before the Multiple Instance loop is started. In addition, it joins these instances before continuing with the remainder of the process (see Figure C.31).

## Solution of WP14 in EKF

This pattern is not supported by the EKF process language, which is mainly caused by the limitation of the EKF process langauge to allow multiple instance of task or sub-process in the process.

## Solution of WP14 in BPEL

The BPEL solution becomes more complex if the number of instances to be created and synchronized is only known at run-time (WP14). A solution in BPEL for this pattern is presented in listing C.20, where a pick activity within a while loop is used, enabling

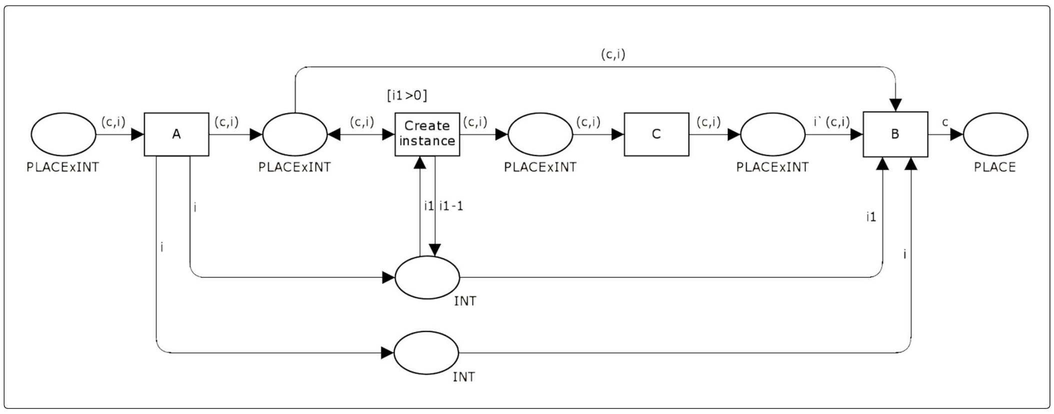


Figure C.31: CPN solution for WP14.

repetitive processing triggered by three different messages: one indication that a new instance is required; one indicating the completion of a previously initiated instance; and one indication that no more instances need to be created. Depending on the received message results eitehr in the invocation of and activity (iteration of the loop) or directly results in the completion of the loop. However, this is only a work-around solution since the logic of this pattern is not directly captured by a BPEL construct. Instead the logic is encoded by means of a loop and a counter, which is incremented each time that a new instance is created, and is decremented each time that an instance is completed. The loop is exited when the value of the counter is zero and no more instances need to be created. We therefore consider this solution to be a partial solution of WP14.

```
<sequence>
 <assign><from>true</from><to>moreInstances</to></from>
 <assign><from>0</from><to>i</to></from>
 <while condition="moreInstances OR i>0">
  <pick>
   <onMessage operation="StartInstance(A)">
   <invoke operation="A"/>
    <assign><from>i+1</from><to>i</to></assign>
   </onMessage>
   <onMessage operation="EndInstance(A)">
    ....
   </onMessage>
   <onMessage NoMoreInstances>
    <assign><from>false</from><to>moreInstnces</to></assign>
   <onMessage>
  </pick>
 </while>
</sequence>
```

Listing C.20: BPEL code for WP14

## C.6 State based patterns

The state based patterns reflect the situations for which the solutions are most easily accomplished through the notion of a state. In this context, we consider the state of the process instance to include the broad collection of data associated with current execution including the state of various tasks, as well as process relevant working data such as the case data elements. The following state based patterns are considered: Deferred Choice (WP16), Interleaved Parallel Routing (WP17) and Milestone (WP18).

### Deferred Choice pattern (WP16)

The deferred choice represents a type of choice, similar to the exclusive choice pattern, however the basis for determining the path that will be taken is different. The exclusive choice pattern is based on the evaluation of process data, while the deferred choice pattern is based on the occurrence of an event. The branch for which the first event occurs is activated and the alternative paths are withdrawn (see Figure C.32).

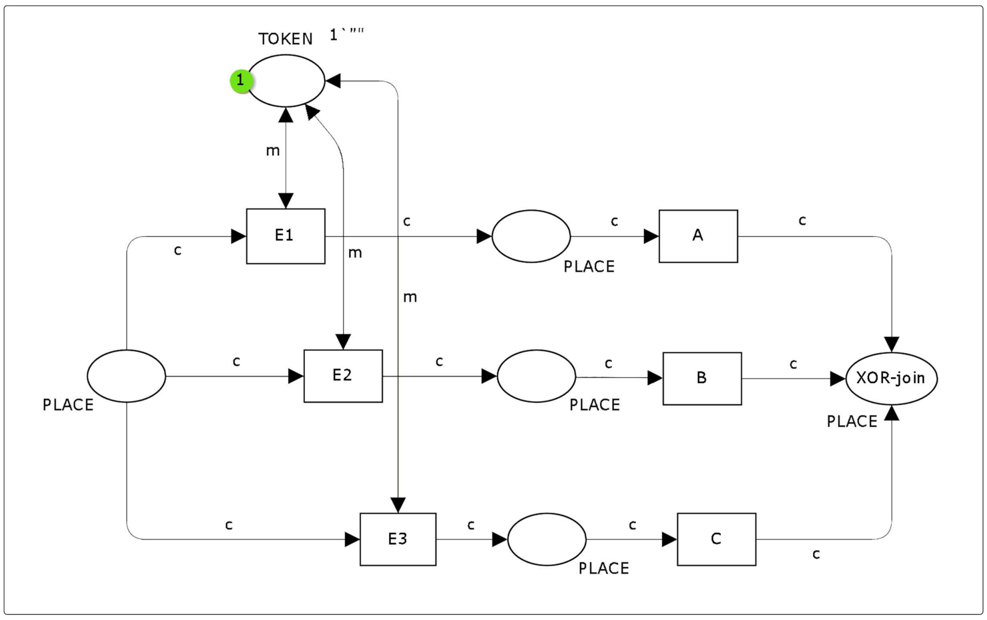
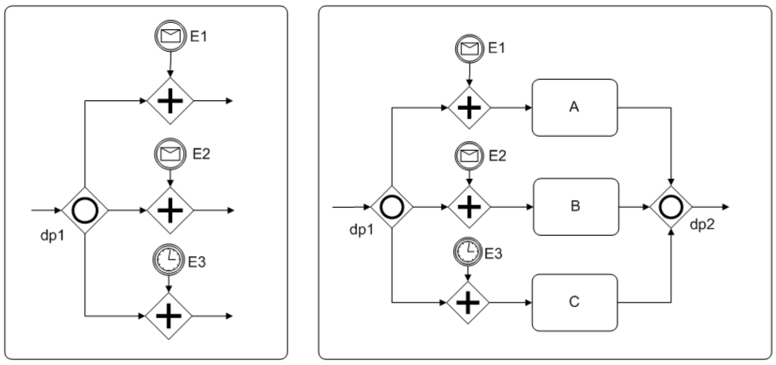


FIGURE C.32: CPN solution for WP16.

### Solution of WP16 in EKF

In the EKF modeling language the deferred choice is modeled as an XOR-split-decision-point, which is contains unconditional sequence flows, which link the XOR-decision-point with two or more intermediate events (see Figure C.33(a)). The first event received by the process engine specifies the continuation of the process and the other paths are withdrawn. In extend to the deferred choice, the structured deferred choice specifies that all branches are eventually merged into a single point (see Figure C.33(b)).



(a) Deferred Choice (b) Structured Deferred Choice

FIGURE C.33: EKF solution for WP16.

### Solution of WP16 in BPEL

A deferred choice is directly supported by the BPEL pick activity as presented in listing C.21.

```
<pick>
 <onMessage operation="E1">
  <invoke operation="A"/>
 </onMessage>
 <onAlarm operation="E2">
  <invoke operation="B"/>
 </onAlarm>
 <onMessage operation="E3">
  <invoke operation="C"/>
 </onMessage>
</pick>
```

LISTING C.21: BPEL code for WP16

### Interleaved Parallel Routing (WP17)

The intent of the interleaved parallel routing pattern is that several tasks are to be performed in sequence (each task is perfromed only once and not in parallel, as the name of the pattern suggests), but the order of execution is arbitrary and not known at design-time (see Figure C.34).

### Solution of WP17 in EKF

In the EKF process this pattern is modeled as a number of tasks, for which the conditional flow specifies that transitions from one task to another (see Figure C.35). This solution still allows that multiple paths from start to end are allowed. This is accomplished through the semantics of the rule engine, as it is allowed to implement rules in such a way that a certain conclusion can only be reached if a previous conclusion has been satisfied. The run-time requirement is also satisfied, because the rule engine is

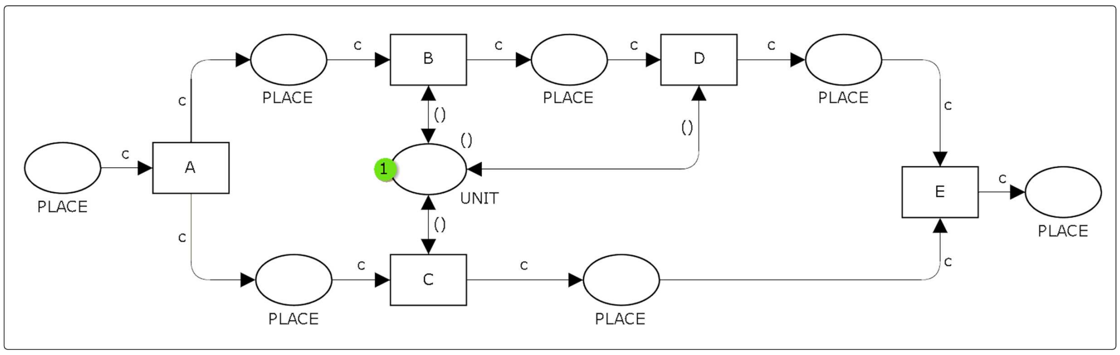


FIGURE C.34: CPN solution for WP17.

responsible to derive conclusions with respect to the pre-condition of a task. Notice that a deadlock can occur, as rules conclusions are not properly aligned with the process. We consider that this pattern is only partially supported by the EKF process language, as an additional rule engine is required to accomplish the semantics of WP17.

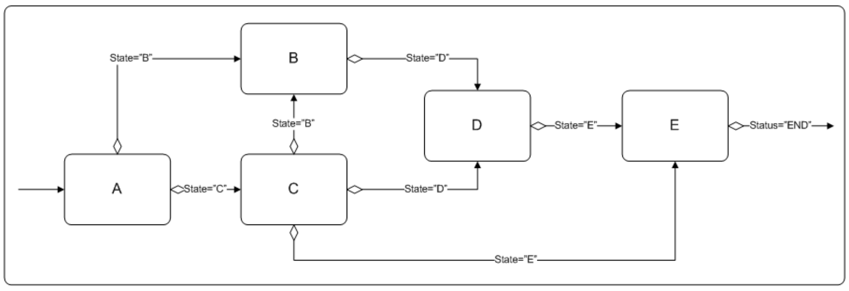


FIGURE C.35: EKF solution for WP17.

## Solution of WP17 in BPEL

The semantics of this pattern can be implemented using the EKF rule engine, by invocation of an operation *decision service* as presented at line 9 and 37 in listing C.22. The result retrieved from the decision is assigned to a BPEL variable state. This variable represents the unique state in the process of the WP17 pattern. A while activity enables the iteration of interleaved parallel routing until the *END* state has been satisfied (end of the interleaved routing). A switch activity nested in a while activity results in the activation of only one task for which pre-condition is satisfied. In extend of the EKF solution, faults in the rule implementation could result in the introduction of unhanded states, which could cause deadlocks. Unhanded states should therefore result in a *condition violated exception* causing the termination of the process through a fault handler. Notice that for this solutions the same limitation as considered for the EKF solution for WP17.

```
1 <scope name="WP17">
2  <faultHandler>
3   <catch faultName="fConditionViolatedException">
4    <invoke operation="Terminate"/>
```

```
   </catch>
  </faultHandler>
  <sequence>
   <!-- Get the state value from the rule engine -->
   <invoke operation="decisionservice" inputvariable="request"
    outputvariable="result"/>
   <assign><from>result</from><to>State</to></assign>
   <!-- repeat until the end state has been reached -->
   <while condition="State=END">
    <sequence>
     <switch>
      <case condition="State=A">
       <invoke operation="A"/>
      </case>
      <case condition="State=B">
       <invoke operation="B"/>
      </case>
      <case condition="State=C">
       <invoke operation="C"/>
      </case>
      <case condition="State=D">
       <invoke operation="D"/>
      </case>
      <case condition="State=E">
       <invoke operation="E"/>
      </case>
      <otherwise>
       <!-- state must be handled otherwise throw exception -->
       <throw faultName="fConditionViolatedException">
      </otherwise>
     </switch>
     <!-- Update the state for the next iteration -->
     <invoke operation="derivedecision" variable="result"/>
     <assign><from>result</from><to>State</to></assign>
    </sequence>
   </while>
  </sequence>
</scope>
```

Listing C.22: BPEL code for WP17

### Milestone (WP18)

The intent of the milestone pattern is that a task can be performed only when a certain milestone has been met and cannot be performed after the milestone expires. The milestone pattern describes the scenario in which a task can be performed multiple times only after the occurrence of an enabling event, but before the occurrence of a disabling event (see Figure C.36).

### Solution of WP18 in EKF

In the EKF process language this pattern can be modeled as a deferred choice in a while loop, such that a task is performed an arbitrary number of times based on the

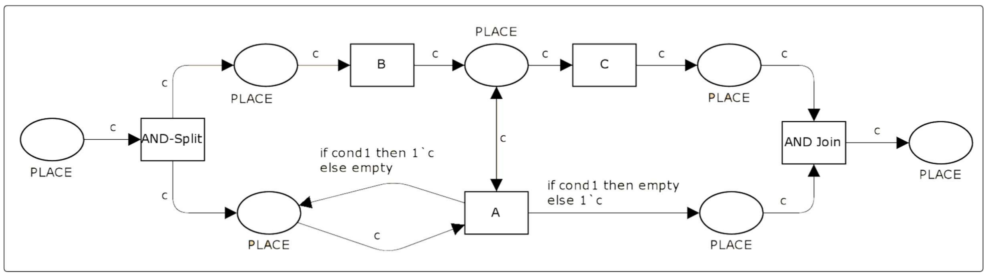


FIGURE C.36: CPN solution for WP18.

evaluation of a post-condition (see Figure C.37). The post-condition of the while loop defines under which conditions the milestone has been met. The deferred choice in the while loop specifies, that the task A can be performed an arbitrary number of times, until the milestone has been satisfied. The limitation of this solution is that the activation of task can not be restricted by any parallel treads, this solutions is therefore considered to be a partial solution for WP18.

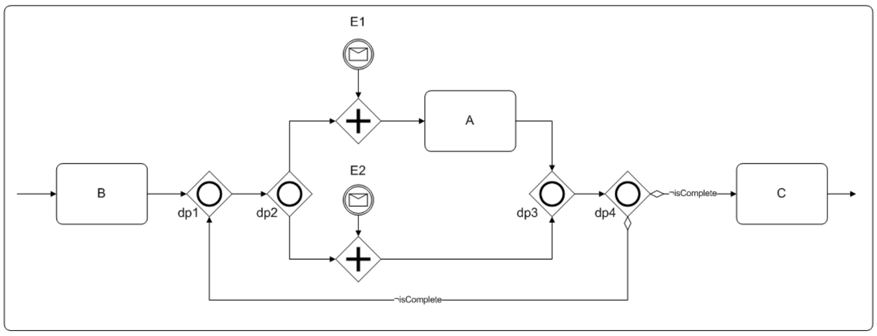


FIGURE C.37: EKF solution for WP18.

### Solution of WP18 in BPEL

In BPEL there does not exist a basic activity for capturing this pattern. We therefore consider a work-around for the BPEL solution based on the EKF solution (see Listing C.23). A deferred choice between execution the task C, or execution task A, is made. A while loop is used to guarantee that as long as C is chosen, A can be execution an arbitrary number of time. Notice that for the BPEL solution the same limitation are considered as defined for EKF solution for WP18.

```
<sequence>
 <invoke operation="B"/>
 <assign><from>true</from><to>isCompleted</to></assign>
 <while condition="isCompleted=true">
  <pick>
   <onMessage operation="E1">
    <invoke operation="A">
   </onMessage>
   <onMessage operation="E2">
    <assign><from>false</from><to>isCompleted</to></assign>
   </onMessage>
  <pick>
 </while>
 <invoke operation="C">
</sequence>
```

LISTING C.23: BPEL code for WP18

## C.7 Cancellation Patterns

Various concepts of exception handling in the process are based on cancellation. The cancellation patterns allow the cancellation of task or a grouping of task through the patterns: Cancel Task (WP19), Cancel Case (WP20) and Cancel Region (WP25).

### Cancel Task (WP19)

The cancel task pattern describes the completion of two parallel task, such that one task signals the completion of the other task. The signal mechanism results in the cancellation of the task being signaled (see Figure C.38).

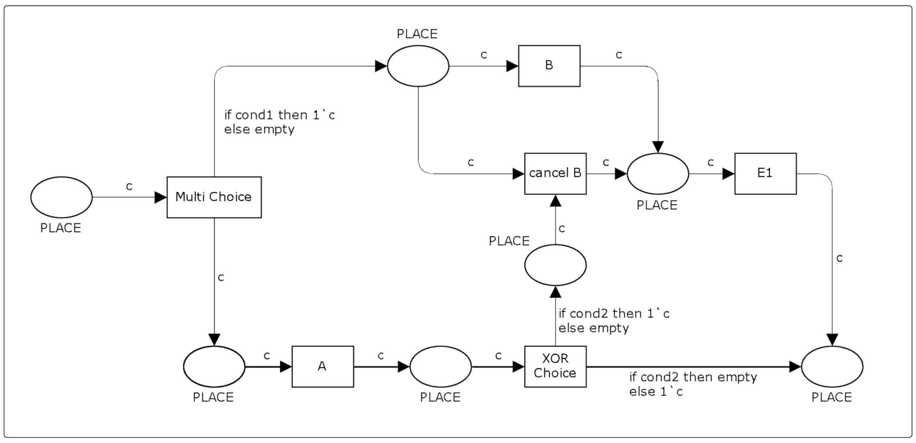


FIGURE C.38: CPN solution for WP19.

### Solution of WP19 in EKF

In the EKF this pattern is implemented as two task activated through and OR-split-decision-point. Each task belongs to a different sub-process, for which the completion of task A results in the cancellation of task instances of the sub-process C. In our example as presented in figure C.39 the cancellation is only performed if the task was activated such that flow condition C2 must be satisfied. In the EKF the condition C2 is derived based on the variables of the sub-process C maintained by the EKF process engine.

### Solution of WP19 in BPEL

In BPEL this pattern can be implemented for two parallel running scope activities, such that the completion of one scope activities throws a fault(s) triggering the completion

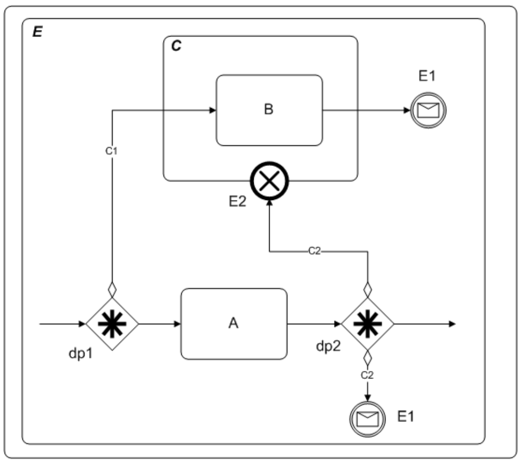


FIGURE C.39: EKF solution for WP19.

of the parallel running scope activity (see Listing C.24). Notice that the fault can only be thrown in case the condition C2 is satisfied. In BPEL the occurrence of the fault causes all instance in the scope to be aborted and continues handling the fault through the corresponding catch. In the fault handler the post-condition of the canceled task is performed and the end event signals the completion of the scope. The BPEL solution of WP19 is considered to be a partial solution as their still remain limitations when translating the sub-process of the EKF process language int the BPEL scope activity.

```
<process name="E">
 <!-- faulthandler required to terminate task t2 in case it was still active -->
 <faultHandlers>
  <catch faultName="Start(E2)">
   <sequence>
    <invoke operation="E1"/>
    <invoke operation="End(C)"/>
   </sequence>
  </catch>
 </faultHandlers>
 <eventHandlers>
 <!-- eventHandler to process task B -->
 <onMessage operation="Start(A)">
  <sequence>
   <invoke operation="A"/>
   <switch>
    <case condition="C2">
     <throw name="Start(E2)"/>
    </case>
    <otherwise>
     <invoke operation="End(C)"/>
    <otherwise>
   </switch>
  </sequence>
 </onMessage>
 <!-- eventHandler to process task B -->
 <onMessage operation="Start(B)">
```

```
  <sequence>
   <invoke operation="B"/>
   <invoke operation="E1"/>
  </sequence>
 </onMessage>
 </eventHandlers>
 <!-- start point of the scope -->
 <flow supressJoinFailure="yes">
  <invoke operation="Start(A)"/>
  <invoke operation="Start(B) joinCondition="C1"/>
 </flow>
</process>
```

LISTING C.24: BPEL code for WP19

### Cancel Case pattern (WP20)

The cancel case pattern is an extension of the cancel task pattern. This pattern allows cancellation of the entire (sub-)process at a certain point in the process. All instances in the process are aborted and the process continues processing an alternative path (see Figure C.40)[2].

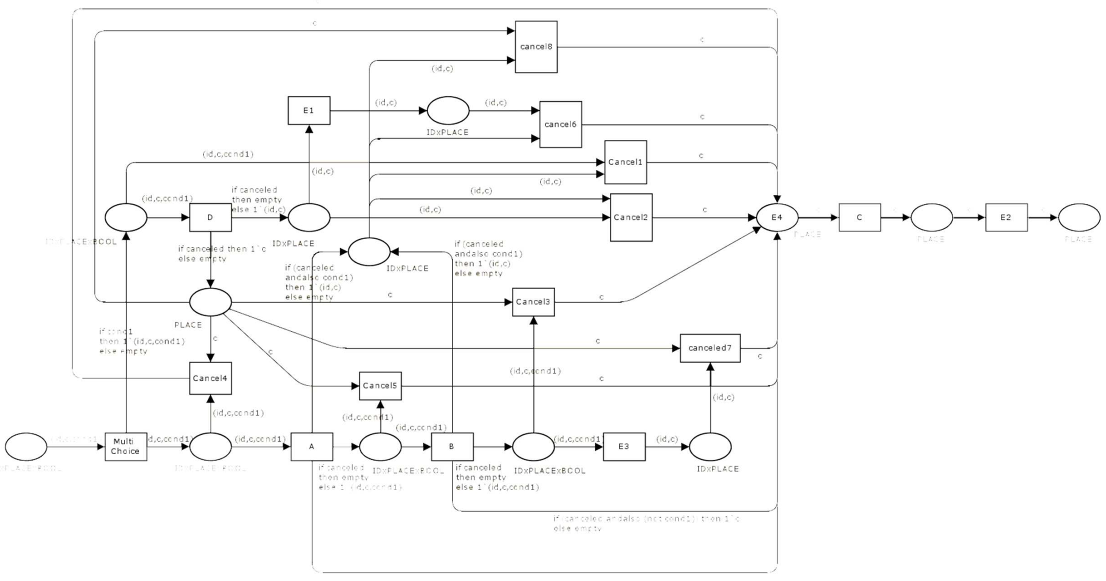

cancel8
E1
IDxPLACE
cancel6
Cancel1
Cancel2
Cancel3
Cancel4
Cancel5
canceled7
D
E4
C
E2
PLACE
Multi Choice
A
B
E3
IDxPLACExBCOL
(id,c)
(id,c,cond1)
c
if canceled then empty else 1`(id,c)
if canceled then 1`c else empty
if (canceled andalso cond1) then 1`(id,c) else empty
if cond1 then 1`(id,c,cond1) else empty
if canceled then empty else 1`(id,c,cond1)
if canceled andalso (not cond1) then 1`c else empty

FIGURE C.40: CPN solution for WP20.

### Solution of WP20 in EKF

In the EKF process language the cancel case pattern is modeled as a task containing an outgoing conditional flow to an intermediate cancel event (see Figure C.41). The cancel event causes the process engine to abort all active instances in the sub-process E and continue processing from the event forward (in our example this is processing task C).

[2]Modeling this pattern in CPN results in a substantial increases of the model complexity, because for all possible combination of tokens which are allowed for cancellation result in an introduction of a transition.

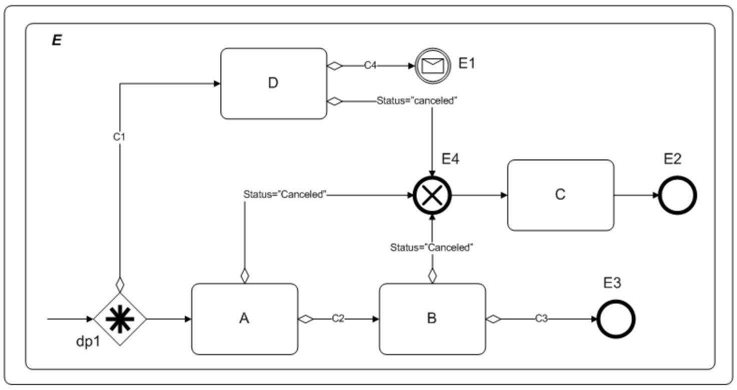


FIGURE C.41: EKF solution for WP20.

### Solution of WP20 in BPEL

In BPEL the cancel case pattern can be implemented as a fault handler, which causes all instances in the process to be terminated and create a new instance which specifies a specific point in the (sub-)process from which the process is continued (see Listing C.25). The fault handler must be defined at the process level, such that its reachable for all scopes in the process. The BPEL solution of WP20 is considered to be a partial solution as their still remain limitations when translating the sub-process of the EKF process language int the BPEL scope activity.

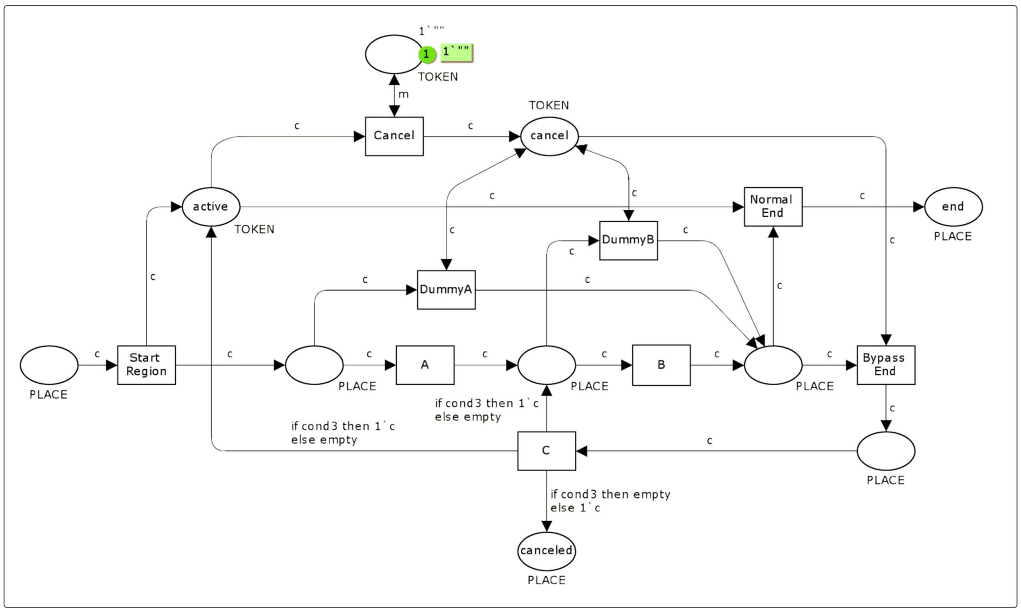
```
<process name="E">
 <faultHandler>
  <!-- eventHandler to process cancellation -->
  <catch faultName="E4">
   <invoke operation="Start(CANCELED)"/>
  </catch>
 </faultHandler>
 <eventHandler>
  <!-- eventHandler to continue after cancellation -->
  <onMessage operation="Start(CANCELED)">
   <sequence>
    <invoke operation="C"/>
    <invoke operation="End(E2E3)"/>
   </sequence>
  </onMessage>
 </eventHandler>
 <!-- scope contained in the process -->
 <scope>
  <eventHandler>
   <!-- eventHandler to process task A -->
    <onMessage operation="start(A)">
     <sequence>
      <invoke operation="A"/>
      <switch>
       <case condition="C2">
        <sequence>
         <invoke operation="B">
         <switch>
```

```
          <case condition="C3">
           <invoke operation="End(E2E3)"/>
          </case>
          <otherwise>
           <throw faultName="Start(E4)"/>
          </otherwise>
         </switch>
        </sequence>
       </case>
       <otherwise>
        <throw fautName="Start(E4)"/>
       </otherwise>
      </switch>
     </sequence>
    </onMessage>
    <!-- eventHandler to process task D -->
    <onMessage operation="Start(D)">
     <sequence>
      <invoke operation="D"/>
      <switch>
       <case condition="C4">
        <invoke operation="E1"/>
       </case>
       <otherwise>
        <throw fautName="Start(E4)"/>
       </otherwise>
      </switch>
     </sequence>
    </onMessage>
   </eventHandler>
   <!-- start point of the scope -->
   <sequence>
    <flow supressJoinFailure="yes">
     <invoke operation="Start(A)"/>
     <invoke operation="Start(D)" joinCondition="C1"/>
    </flow>
    <receive operation="End(E2E3)">
    </sequence>
  </scope>
</process>
```

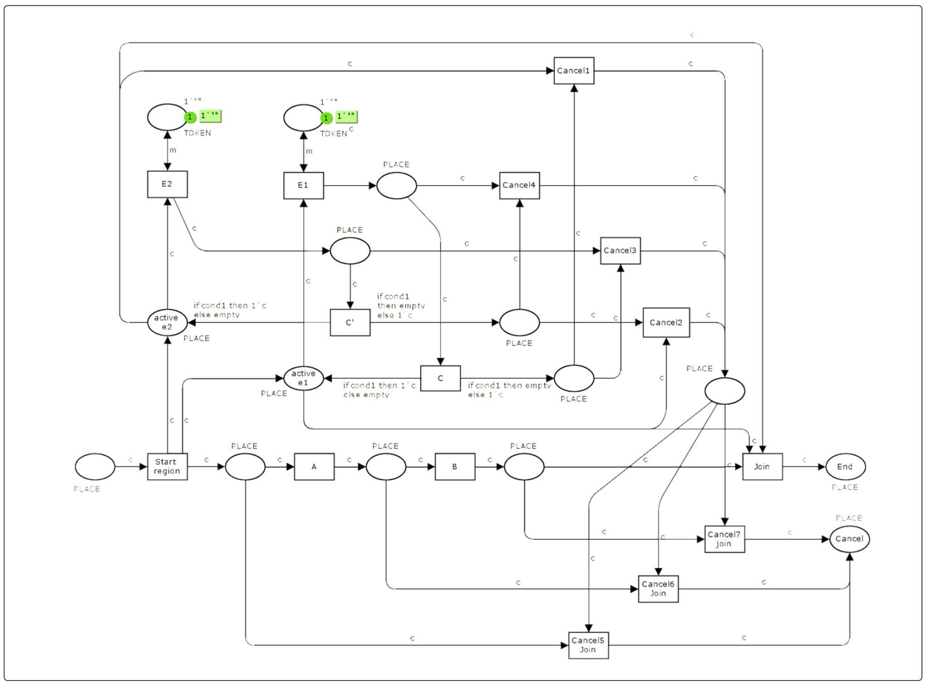
Listing C.25: BPEL code for WP20

### Cancel region (WP25)

The cancel region pattern allows the disabling of a grouping of tasks in the process. Any of the tasks which are enabled at the moment of cancellation are withdrawn. The cancel region is the option of being able to cancel a series of (potentially unrelated) tasks is a useful capability, particularly for handling unexpected errors or for implementing forms of exception handling. Two distinct cancel region approaches are supported by the EKF process language: *cancellation through trigger event* (a) (see Figure C.42(a)) and *cancellation through exception* (b) (see Figure C.42(b))[3]. The cancellation through the event trigger causes all tokes in the region to be removed and continues processing a single token through a bypass in the process. The cancellation through the exception event does not remove any tokens at the occurrence of the exception, instead it creates a new instance to handle the exception.



(a) Cancel Region through trigger event



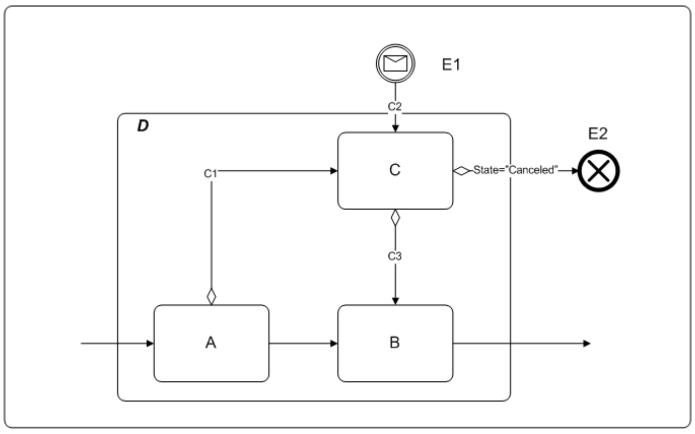
(b) Cancel Region through exception

FIGURE C.42: CPN solutions for WP25.

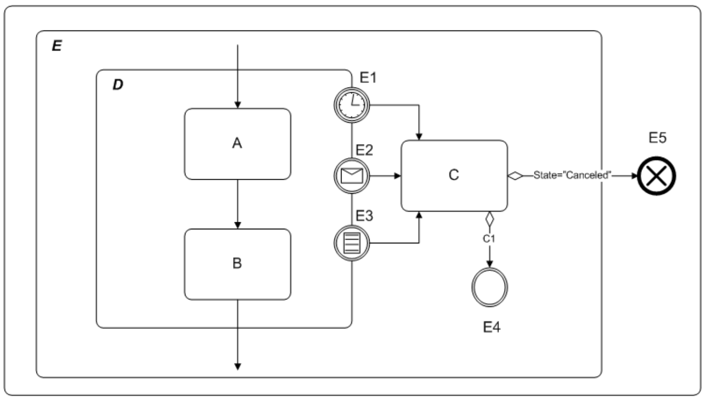
[3]Notice the complexity of the CPN model increases as the cancellation of each token, which must be withdrawn requires the introduction of a transition.

### Solution of WP25 in EKF

Both cancellation strategies are supported by the EKF, where the grouping of tasks is represented by a sub-process. The first alternative (a) (see Figure C.43(b)) specifies an interrupt trigger event, which causes all instances in the sub-process to be aborted and continues processing from the event forward. The EKF process language allows constraints on the trigger event, such that the trigger event is only accepted in case the condition is satisfied. In the second alternative (b) (see Figure C.43(b)) one or more exceptions can be defined for a sub-process. The exception does not affect the processing of the sub-process, but creates a new instance to process task C (therefore running in parallel with internal processing of sub-process D). The completion of task C can either result in implicit termination (through the NOP event E4) or explicit cancellation of the sub-process (through cancel event E5).



(a) Cancel Region through trigger event



(b) Cancel Region through exception

FIGURE C.43: EKF solutions for WP25.

### Solution of WP25 in BPEL

The BPEL solution for alternative (a) can be implemented as an (message) event handler, which throws a fault (see Listing C.26). The fault handler is required, because the on the occurrence of the event all instances of the scope activity must be withdrawn. The BPEL solution of WP25(a) is considered to be a partial solution as their still remain limitations when translating the sub-process of the EKF process language int the BPEL scope activity.

```
<scope name="E">
 <!-- faulthandler required to terminate sub-process D-->
 <faultHandlers>
  <catch faultName="Fault(E1)">
   <!-- preceding after the occurrence of trigger event E1 -->
   <invoke operation="Start(C)"/>
  </catch>
 </faultHandlers>
 <scope name="D">
  <eventHandler>
   <onMessage operation="E1">
    <!-- Only process trigger event E1 if C2 is satisfied -->
    <switch>
     <case condition="C2">
      <throw faultName="Cancel(E1)"/>
     </case>
     <otherwise>
      <empty/>
     </otherwise>
    </switch>
   </onMessage>
   <onMessage operation="Start(B)">
    <sequence>
     <invoke operation="B"/>
     <!-- preceding tasks after completion of B -->
     <invoke operation="End(D)"/>
    </sequence>
   </onMessage>
   <onMessage operation="Start(C)">
    <sequence>
     <invoke operation="C"/>
     <switch>
      <case condition="C3">
       <invoke operation="Start(B)"/>
      </case>
      <otherwise>
       <throw faultName="Cancel(E2)"/>
      </otherwise>
     </switch>
    </onMessage>
   </eventHandler>
   <!-- start point of the the scope E -->
   <sequence>
   <invoke operation="A">
   <switch>
```

```
    <case condition="C1">
   <invoke operation="Start(B)">
    </case>
    <otherwise>
   <invoke operation="Start(C)">
    </otherwise>
   </switch>
   <receive operation="End(D)"/>
  </sequence>
 </scope>
</scope>
```

Listing C.26: BPEL code for WP25(a)

The BPEL the cancellation exception is implemented as an onMessage or onAlarm event handler, creating a new instance in the process (see Listing C.27). The active instance in the sub-process remains active unless the process is explicitly canceled. The BPEL solution of WP25(b) is considered to be a partial solution as their still remain limitations when translating the sub-process of the EKF process language int the BPEL scope activity.

```
<scope name="E">
 <!-- eventHandler for scope E-->
 <eventHandler>
  <onMessage operation="Start(C)">
 <sequence>
  <invoke operation="C"/>
  <switch>
   <case condition="C1">
  </empty>
   </case>
   <otherwise>
  <throw faultName="Cancel(E5)"/>
   </otherwise>
  </switch>
  </onMessage>
 </eventHandler>
 <scope name="D">
  <!-- eventHandler for scope D -->
  <eventHandler>
   <onAlarm name="E1">
  <invoke operation="Start(C)"/>
   </onAlarm>
   <onMessage operation="E2">
  <invoke operation="Start(C)"/>
   </onMessage>
   <onMessage operation="E3">
  <invoke operation="Start(C)"/>
   </onMessage>
  </eventHandler>
  <!-- processing of scope D -->
  <sequence>
    <invoke operation="A"/>
    <invoke operation="B"/>
  </sequence>
 </scope>
</scope>
```

Listing C.27: BPEL code for WP25(b)

## C.8 Termination Patterns

The termination patterns deal within the circumstance under which a process is considered to be completed. The following termination patterns are considered: Implicit

Termination (WP11) and Explicit Termination (WP43).

### Implicit termination (WP11)

The implicit termination pattern allows that a specific path of a process is completed without other parallel path to be required to end as well. A given process instance should be terminated when there are no remaining work items that are able to be done now or at any time in the future without being in deadlock. There is an objective means of determine that the process instance has successfully completed. The implicit termination is more or less a relaxation of the design rules: the process complete when the tasks on each of the branches complete. The implicit termination is not modeled explicitly in modeled CPN, as by definition *'all tokens must end in a single end place'*.

### Solution of WP11 in EKF

In the EKF process language the implicit termination can be modeled as multiple notification event(s) or NOP event(s) at the end point (see Figure C.44). Notice that the (sub-)process is only considered to be completed if all active branches (e.g. notification message events are send or NOP event is reached) have reached their completion.

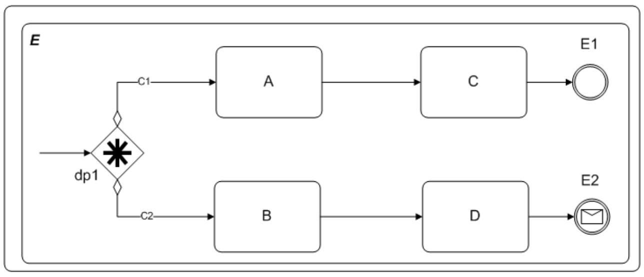


FIGURE C.44: EKF solution for WP11.

### Solution of WP11 in BPEL

The implicit termination is supported by the BPEL flow construct and links (see Listing C.28). This allows that a scope can have multiple sink activities (i.e. activities not being a source of any link) without requiring one unique termination activity. The attribute *suppressJoinFailuire* is assigned to *no*, otherwise the implicit termination could be result in termination when a deadlock has occurred.

```
<scope name="E">
 <flow suppressJoinFailure="no">
  <links>
   <link name="AC"/>
   <link name="BD"/>
   <link name="DE2"/>
  <links>
  <invoke operation="A" joinCondition="C1">
   <source linkName="AC"/>
  </invoke>
  <invoke operation="B" joinCondition="C2">
```

```
    <source linkName="CD"/>
   </invoke>
   <invoke operation="C">
    <target linkName="AC"/>
   </invoke>
   <invoke operation="D">
    <source linkName="DE2"/>
    <target linkName="BD"/>
   </invoke>
   <invoke operation="E2">
    <target linkName="DE2"/>
   </invoke>
  </flow>
</scope>
```

Listing C.28: BPEL code for WP11

### Explicit termination pattern (WP43)

A given (sub-)process instance should terminate when it reaches a nominated state. Typically this is denoted by a specific end node. When this end node is reached, any remaining work in the process is canceled and the overall process instances is recorded as having completed successfully, regardless of whether there are any task in progress or remaining to be executed. In CPN the end state is explicitly modeled by means on a single end place.

### Solution of WP43 in EKF

The EKF process language allows two ways to model this pattern: first, the termination event causes the direct termination of all instances in the processes; or an end event specifies the completion of the process under normal conditions (see Figure C.45).

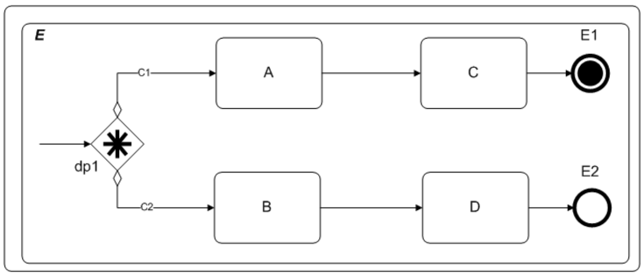


Figure C.45: EKF solution for WP43.

### Solution of WP43 in BPEL

In BPEL the explicit termination is modeled as a process for which no more activity instances remain to complete or and explicitly through the terminate activity. In our example the flow can activate multiple branches of activities, such that the completion of the first branch directly results in the completion of the process. The termination event directly triggers the completion of the process (see Listing C.29).

```
<process name="E">
 <eventHandler>
  <onMessage operation="End(E1)">
   <sequence>
    <invoke operation="E1"/>
    <terminate name="E1"/>
   </sequence>
  </onMessage>
  <onMessage operation="End(E2)">
   <sequence>
    <invoke operation="E2"/>
    <terminate name="E2"/>
   </sequence>
  </onMessage>
 </eventHandler>
 ...
 <flow suppressJoinFailure="yes">
  <links>
   <link name="AC"/>
   <link name="BD"/>
   <link name="CE1"/>
   <link name="DE2"/>
  </links>
  <invoke operation="A" joinCondition="C1">
   <source linkName="AC"/>
  </invoke>
  <invoke operation="B" joinCondition="C2">
   <source linkName="BD"/>
  </invoke>
  <invoke operation="C">
   <source linkName="CE1"/>
   <target linkName="AC"/>
  </invoke>
  <invoke operation="D">
   <source linkName="DE2"/>
   <target linkName="BD"/>
  </invoke>
  <invoke operation="End(E1)">
   <target linkName="CE1"/>
  </invoke>
  <invoke operation="End(E2)">
   <target linkName="DE2"/>
  </invoke>
 </flow>
</process>
```

Listing C.29: BPEL code for WP43

Appendix D

# Case Study: The Everest Mortgage Process

In this case study we aim at presenting a complete EKF process model from the Everest practice and aim at providing a complete and correct transformation of this process model into BPEL code. This case study is therefore composed of a description of the case study (see Section D.1) and a detailed description of how the actual transformation was performed (see Section D.2).

## D.1 Case study explained

The case study we present comes directly from the Everest practice and describes a generic mortgage process from sale to delivery. First, we provide the domain model of the mortgage process (see Section D.1.1), followed by an example of a rule model (see Section D.1.2). Second, we describe the mortgage process and sub-processes from sale to delivery (see Section D.1.3).

### D.1.1 Mortgage domain model example

Figure D.1 gives an example of a mortgage domain (domain model of the mortgage case study) describing the entities and attributes and structural relations of the mortgage process.

This process specifies three type of *actors*:

- A *customer*, which requests for a mortgage loan.

- A *mortgage broker* is the channel through which the loans are sold.
- A *lender* provides mortgage products, which can be sold by the brokers.

Different type of mortgage products could be provided by the lenders. The following mortgage products are considered in this example:

- *Linear Mortgage* (ProductA) both the interest and payments to the property are made. The interest rate decreases (graduated calculation of interest) in time, because the payment amount decreases.
- *Level Payment Mortgage* (ProductB) is similar to the linear mortgage, but defines a constant monthly payment.
- *Savings Mortgage*[1] (ProductC) only requires the payment of the interest, such that no mortgage payments have to be made for the property. Certain insurances must cover the increased risk factors.

A customer applies a mortgage product. The broker gathers the required information about the customer and the mortgage product in a *MortgageRequest*. A request is formalized into an *Offer*, specifying the details of the Mortgage Request. The offer in combination with the *mortgage product* is the outline from which the actual *loan* is derived. From one mortgage product, different loans can be derived, based on the properties from the mortgage request. Before the loan can be provided, a risk analysis is performed, and the required *documents* are gathered (e.g. bank statements, pay stubs, employment registration, lending information and insurance papers, etc). Based on this information a final decision about the loan can be made. Notice that various integrity rules are include to guard the consistency of the domain data.

Following the domain transformation, we can translate the instances of the domain model (input and output of the task) into XSDs, such that it can be used in the BPEL process.

[1]Typical Dutch mortgage product

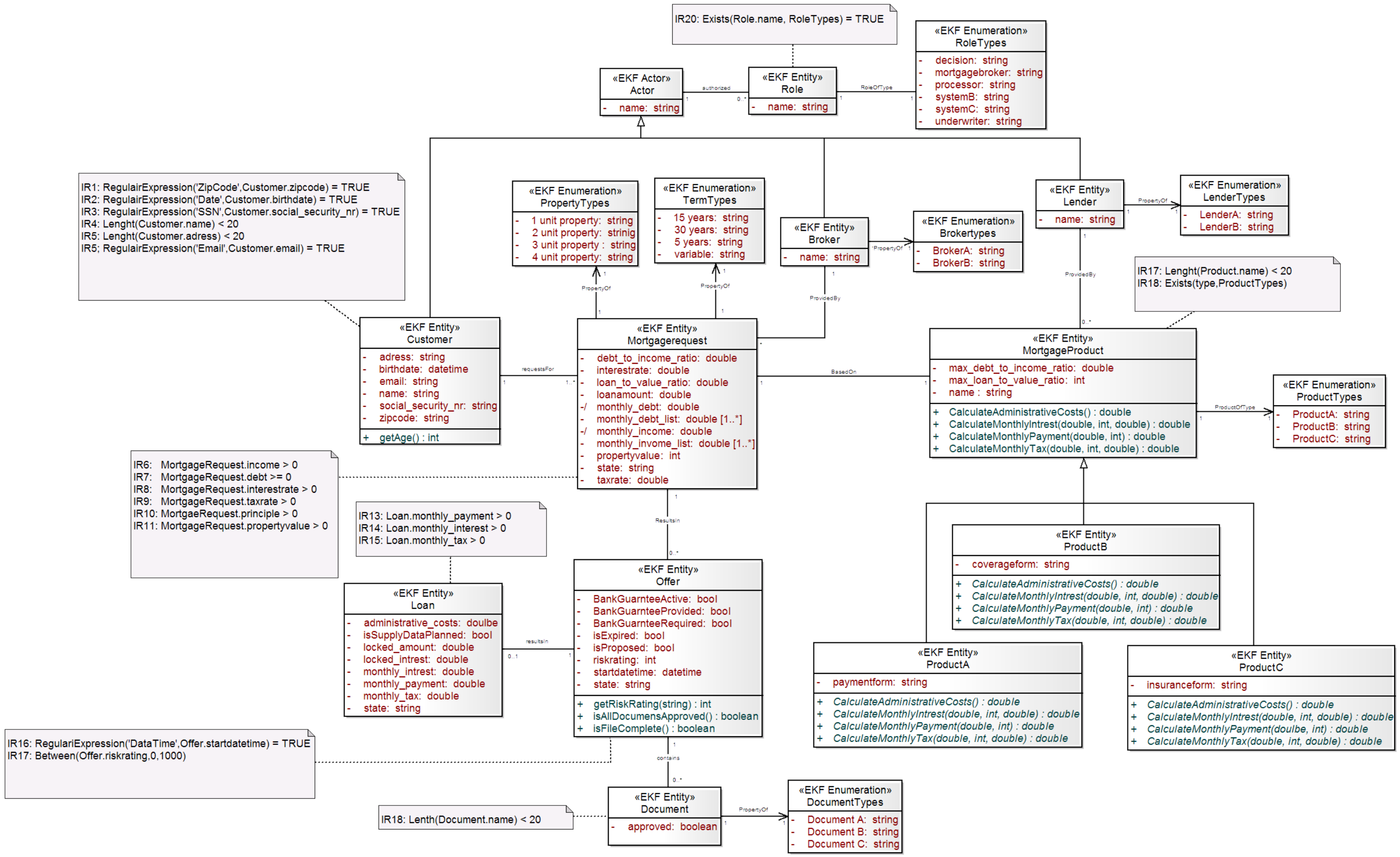
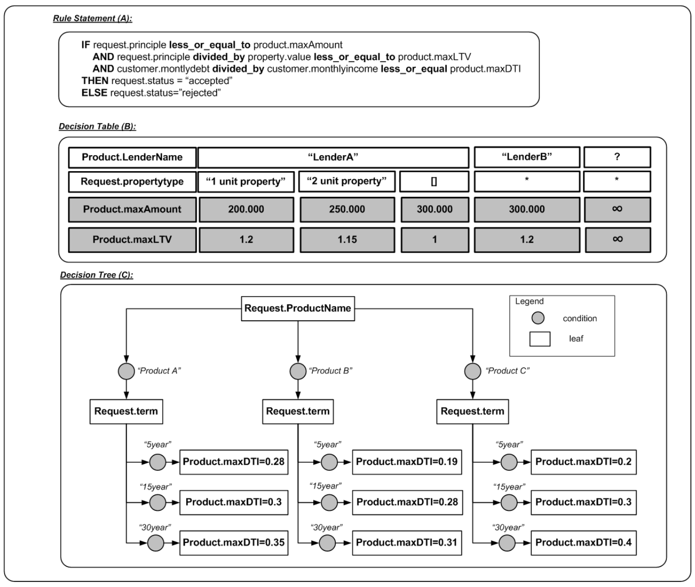

IR20: Exists(Role.name, RoleTypes) = TRUE
«EKF Actor» Actor
- name: string
authorized
1
0..*
«EKF Entity» Role
- name: string
RoleOfType
«EKF Enumeration» RoleTypes
- decision: string
- mortgagebroker: string
- processor: string
- systemB: string
- systemC: string
- underwriter: string
IR1: RegulairExpression('ZipCode',Customer.zipcode) = TRUE
IR2: RegulairExpression('Date',Customer.birthdate) = TRUE
IR3: RegulairExpression('SSN',Customer.social_security_nr) = TRUE
IR4: Lenght(Customer.name) < 20
IR5: Lenght(Customer.adress) < 20
IR5; RegulairExpression('Email',Customer.email) = TRUE
«EKF Enumeration» PropertyTypes
- 1 unit property: string
- 2 unit property: strinig
- 3 unit property : string
- 4 unit property: string
«EKF Enumeration» TermTypes
- 15 years: string
- 30 years: string
- 5 years: string
- variable: string
«EKF Entity» Broker
- name: string
PropertyOf
«EKF Enumeration» Brokertypes
- BrokerA: string
- BrokerB: string
«EKF Entity» Lender
- name: string
PropertyOf
«EKF Enumeration» LenderTypes
- LenderA: string
- LenderB: string
IR17: Lenght(Product.name) < 20
IR18: Exists(type,ProductTypes)
PropertyOf
PropertyOf
ProvidedBy
ProvidedBy
«EKF Entity» Customer
- adress: string
- birthdate: datetime
- email: string
- name: string
- social_security_nr: string
- zipcode: string
+ getAge() : int
requestsFor
1..*
«EKF Entity» Mortgagerequest
- debt_to_income_ratio: double
- interestrate: double
- loan_to_value_ratio: double
- loanamount: double
-/ monthly_debt: double
- monthly_debt_list: double [1..*]
-/ monthly_income: double
- monthly_invome_list: double [1..*]
- propertyvalue: int
- state: string
- taxrate: double
BasedOn
«EKF Entity» MortgageProduct
- max_debt_to_income_ratio: double
- max_loan_to_value_ratio: int
- name : string
+ CalculateAdministrativeCosts() : double
+ CalculateMonthlyIntrest(double, int, double) : double
+ CalculateMonthlyPayment(double, int) : double
+ CalculateMonthlyTax(double, int, double) : double
ProductOfType
«EKF Enumeration» ProductTypes
- ProductA: string
- ProductB: string
- ProductC: string
IR6: MortgageRequest.income > 0
IR7: MortgageRequest.debt >= 0
IR8: MortgageRequest.interestrate > 0
IR9: MortgageRequest.taxrate > 0
IR10: MortgaeRequest.principle > 0
IR11: MortgageRequest.propertyvalue > 0
IR13: Loan.monthly_payment > 0
IR14: Loan.monthly_interest > 0
IR15: Loan.monthly_tax > 0
ResultsIn
0..*
«EKF Entity» ProductB
- coverageform: string
+ CalculateAdministrativeCosts() : double
+ CalculateMonthlyIntrest(double, int, double) : double
+ CalculateMonthlyPayment(double, int) : double
+ CalculateMonthlyTax(double, int, double) : double
«EKF Entity» Loan
- administrative_costs: doulbe
- isSupplyDataPlanned: bool
- locked_amount: double
- locked_intrest: double
- monthly_intrest: double
- monthly_payment: double
- monthly_tax: double
- state: string
resultsIn
0..1
«EKF Entity» Offer
- BankGuarnteeActive: bool
- BankGuarnteeProvided: bool
- BankGuarnteeRequired: bool
- isExpired: bool
- isProposed: bool
- riskrating: int
- startdatetime: datetime
- state: string
+ getRiskRating(string) : int
+ isAllDocumensApproved() : boolean
+ isFileComplete() : boolean
«EKF Entity» ProductA
- paymentform: string
+ CalculateAdministrativeCosts() : double
+ CalculateMonthlyIntrest(double, int, double) : double
+ CalculateMonthlyPayment(double, int) : double
+ CalculateMonthlyTax(double, int, double) : double
«EKF Entity» ProductC
- insuranceform: string
+ CalculateAdministrativeCosts() : double
+ CalculateMonthlyIntrest(double, int, double) : double
+ CalculateMonthlyPayment(doulbe, int) : double
+ CalculateMonthlyTax(double, int, double) : double
IR16: RegulariExpression('DataTime',Offer.startdatetime) = TRUE
IR17: Between(Offer.riskrating,0,1000)
contains
0..*
IR18: Lenth(Document.name) < 20
«EKF Entity» Document
- approved: boolean
PropertyOf
«EKF Enumeration» DocumentTypes
- Document A: string
- Document B: string
- Document C: string

Figure D.1: Example of domain data model of the EKF mortgage process.

### D.1.2 Mortgage rule model example

The examples of the rule representations are presented in figure D.2 and is based on the entities and attributes from the domain model example as presented in the previous section. This rule model specifies the rules which must be considered to determine under which conditions a mortgage request is accepted or rejected as the main objective of the task *qualify request*. To accomplish this objective this task is decomposed into the following requirements, from which the rules can be modeled:

- The loan request may not exceed the maximal loan amount as specified for a certain loan product.
- The Loan to Value ratio (LTV) calculated by dividing the requested loan amount by the value of the property may not exceed the maximal LTV for a certain mortgage product.
- The Debt to Income ratio (DTI) calculated by dividing the monthly debt of the customer by the monthly income of the customer may not exceed the maximal DTI for a certain mortgage product.
- The maximal product amount and maximal LTV are variable for a mortgage request and is based on the type of mortgage property.
- The maximal DTI is variable for a mortgage request and is based on the type of product and term (lead time) for which the product is requested.



***Rule Statement (A):***

```
IF request.principle less_or_equal_to product.maxAmount
   AND request.principle divided_by property.value less_or_equal_to product.maxLTV
   AND customer.montlydebt divided_by customer.monthlyincome less_or_equal product.maxDTI
THEN request.status = “accepted”
ELSE request.status=”rejected”
```

***Decision Table (B):***

| Product.LenderName | “LenderA” | | | “LenderB” | ? |
|---|---|---|---|---|---|
| Request.propertytype | “1 unit property” | “2 unit property” | [] | * | * |
| Product.maxAmount | 200.000 | 250.000 | 300.000 | 300.000 | ∞ |
| Product.maxLTV | 1.2 | 1.15 | 1 | 1.2 | ∞ |

***Decision Tree (C):***

FIGURE D.2: Example of rule representations in EKF: rule statement (A), decision table (B), decision tree (C).

### D.1.3 Mortgage process example

The mortgage process is composed of generally three sub-processes: application, processing, underwriting and bankguarantee (see Figure D.3). The mortgage process starts on the reception of either a requestA or requestB. For a *requestA* the information is imported from an external system and for *requestB* the information is registered through the EKF portal application. In the sub-process *application* the information about the customer and mortgage products is gathered and validated. Mortgage products specify the outline (composed of business rules) which is used to derive the actual loan. A request is formalized into an offer, specifying the details of the actual mortgage loan. From one mortgage product, different loans can be derived, based on the properties from the mortgage request. In the sub-process *process* a risk analysis and credit analysis is performed and the documents are gathered (e.g. bank statements, pay stubs, employment registration, lending information and insurance papers etc). Based on the information from the processing a final decision and administration of the loan is performed in the sub-process *underwriting.*

The *cancellation* process specifies the proceeding of the process in case the mortgage request is canceled. After cancellation the process can either be reactivated (manually) through a message event. Reactivation either results in processing of an existing offer (if it was not expired and an offer was already proposed) or the creation of a new application request. Reactivation is only possible within the reactivation period, such that when this period expires the entire process is closed and terminated.

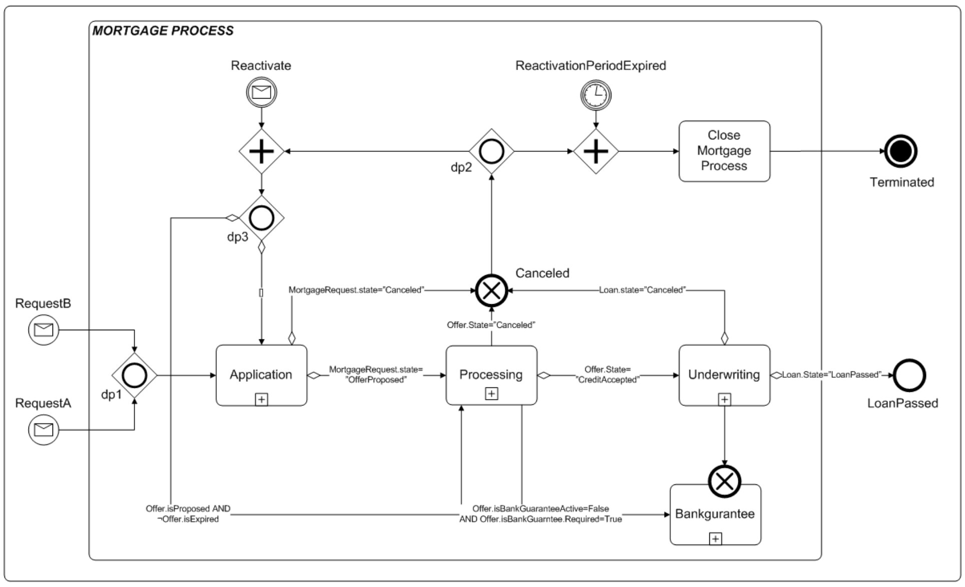

MORTGAGE PROCESS
Reactivate
ReactivationPeriodExpired
Close Mortgage Process
dp2
Terminated
dp3
Canceled
MortgageRequest.state="Canceled"
Loan.state="Canceled"
Offer.State="Canceled"
RequestB
RequestA
dp1
Application
MortgageRequest.state= "OfferProposed"
Processing
Offer.State= "CreditAccepted"
Underwriting
Loan.State="LoanPassed"
LoanPassed
Offer.isProposed AND ¬Offer.isExpired
Offer.isBankGuaranteeActive=False AND Offer.isBankGuarntee.Required=True
Bankgurantee

Figure D.3: Mortgage process example.

#### D.1.3.1 sub-process application

In the sub-process *application* (see Figure D.4) the information of the customer and product selection is either manually or automatically registered. The information is validated based on its correctness and completeness and derivation rules are applied to enrich the process data (take for example the pricing of the loan product is calculated based on the information of the request). In case the information was in correct or incomplete it can be manually be updated and changed in the task complement request. If all information is complete the request is qualified, for which it is determined if the customer is eligible to acquire the requested loan product (see rule example). Rejected loan qualifications require manual judgment and accepted request directly results in the proposal of an offer.

The application can be canceled after the completion of the tasks: register requests, complement request or judge request. Alternatively the application can be canceled through the cancel triggered, but only if the pre-condition is satisfied.

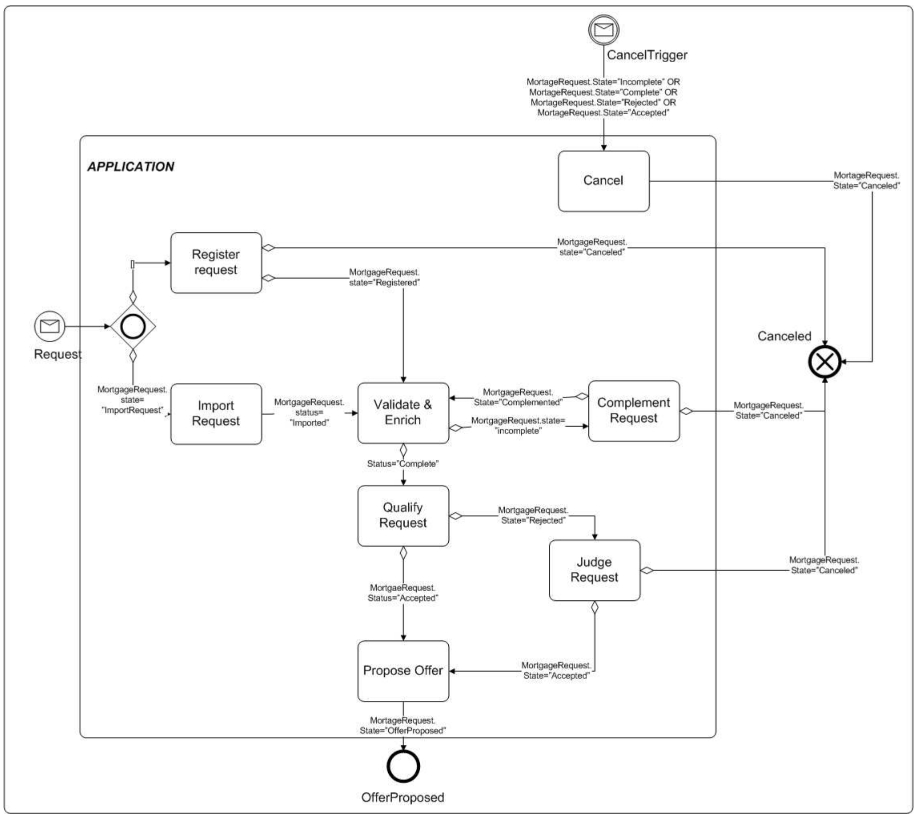


Figure D.4: Application sub-process example.

#### D.1.3.2 sub-process processing

In the sub-process *processing* (see Figure D.5) is started if the proposed offer was accepted by the customer, by returning a signed copy of the offer. Accepting the offer results in the lock of the loan product, which is more or less an agreement of a fixed product interest rate. Notice that the customer must return the offer within a predefined signing period and offer may not violate the general government law. After the customer acceptance the lender personnel also known as *processors* gather the required information and legal documents, which are used to make a decision about the acceptance of the loan. Notice that documents can be included to the file until the file is considered to be complete. In the task update file additional information is gathered (e.g. risk information, credit information) in preparation for the underwriter, to make a final decision about the provision of the loan. This type of data is mostly requested through different channels (e.g. central bureau of credit, local authority, etc.). The completed file is validated for completeness, correctness and possible fraud situations. For every expected fraud manual judgment of the processor is required. Finally the entire loan is judged by the processor resulting in the requirement of additional documents or the acceptance of the credit. Notice that independent of the acceptance of the credit a bank guarantee could be required.

During the sub-process *offer processing* two exceptions are considered: in case the offer expires or almost expires the process can extend the sign period through the task extend sign period or cancel the process. Cancellation is provided through the cancel trigger, but only if the pre-ondition is satisfied.

#### D.1.3.3 sub-process underwriting and bankguarantee

the sub-process *underwriting* (see Figure D.6) is composed of three sub-processes: bankacceptance, supplyloan and bankguarantee. The sub-process bankguarantee is activated independent from the sub-process underwriting, but needs to synchronize at a certain point in the process.

In the sub-process *bankacceptance* the actor underwriter makes a final decision on the approval of the loan, based on the information gathered during the processing. In the sub-process underwriting it is important to identify under which conditions a loan was granted or rejected. Automated rejected loans require manual judgment, from which the loans are finally rejected, resulting in cancellation; or accepted, resulting in the activation of the task complete bank acceptance. The active instances of a bankguarantee will be canceled in case they are still active after the completion of the sub-process bankacceptance.

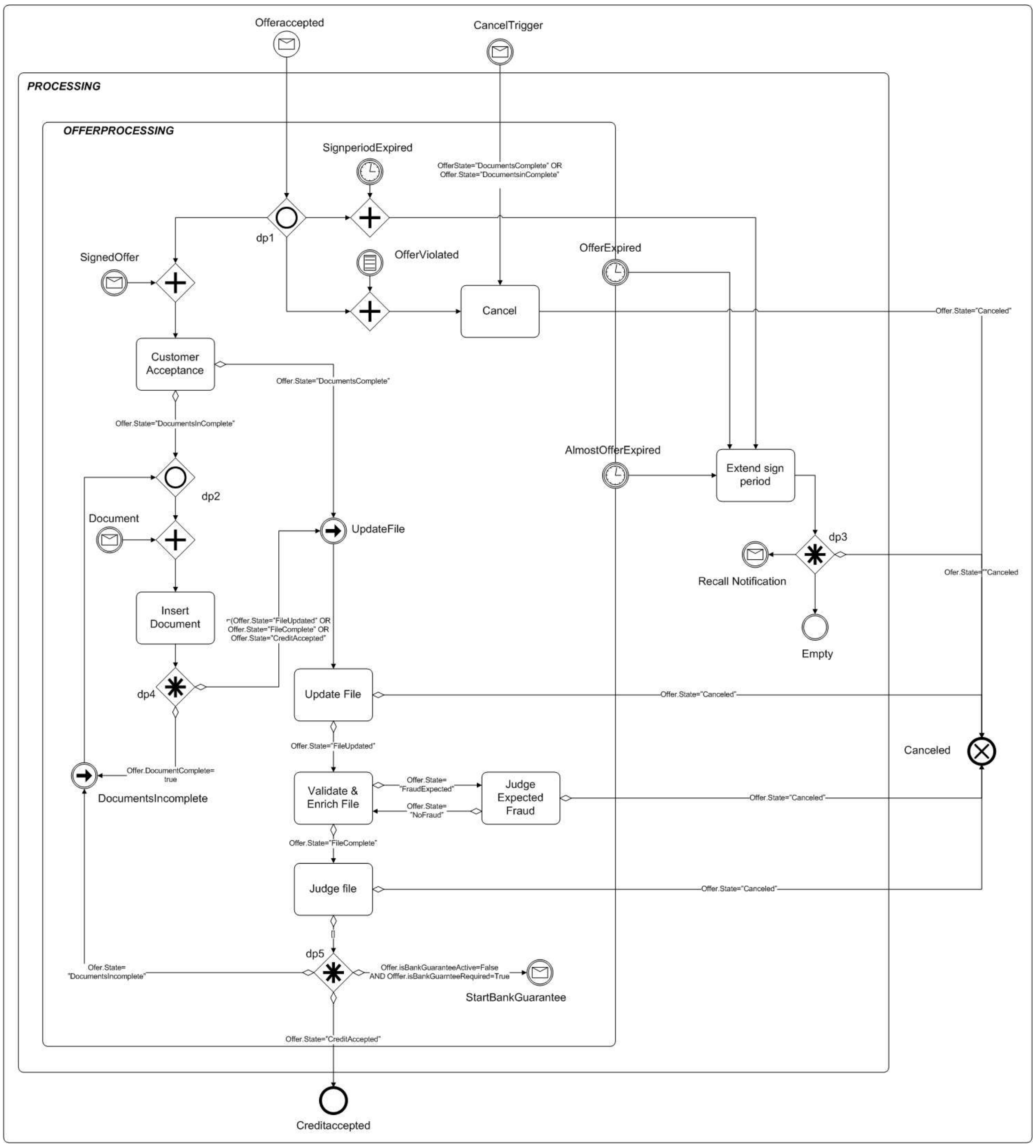


FIGURE D.5: Processing sub-process example.

The sub-process *supplyloan* the loan is accepted by the underwriters and is prepared to be supplied to the customer. In case the supply date is known the supply date can directly be registered, otherwise the notary instructions must be send through an external channel and the process waits until it receives a signed notary contract. The notary contract includes the supply date such that it can be registered. After registration of the supply date the loan is transferred to the administrative system and the mortgage process is completed (marked as loan passed).

The sub-process *bankguarantee* allows the creation and judgment of a bankguarantee. A bank guarantee is only required in certain circumstances, as specified in the sub-process processing and can operate in parallel with the bank acceptance sub-process. Notice that a bank guarantee must be either provided or canceled if the bank acceptance was completed.

Cancellation of the sub-process underwriting is only allowed for the sub-processes bankacceptance and bankguarantee, because during the sub-process supplyloan the customer there exist a certain commitment to the loan. Cancellation is provided through the cancel trigger, but only if the pre-condition is satisfied (due to the commitment of the loan).

## D.2 Transformation of the EKF process into BPEL

In this section we follow the approach as presented in chapter 6, to provide a complete transformation of the EKF mortgage process into BPEL code. We do not consider the first and second step of our transformation approach as these step are considered to be straightforward.

### D.2.1 Transform the the sub-process application into BPEL

**STEP3: Decompose the sub-process application into components**
Following the thrid step of our transformation approach we can identify the following component for the sub-process application (see Figure D.7):

- $C_1$, is translated into an maximal component composed of an exclusive choice and the start point of the sub-process application and either results in the activation of component $C_2$ or cancellation of the process (from line 26 to 50 in Listing D.1).
- $C_2$, is translated into a maximal component composed of exclusive choice components, resulting either in the cancellation or activation of $C_3$ (from line 51 to 95 in Listing D.1).

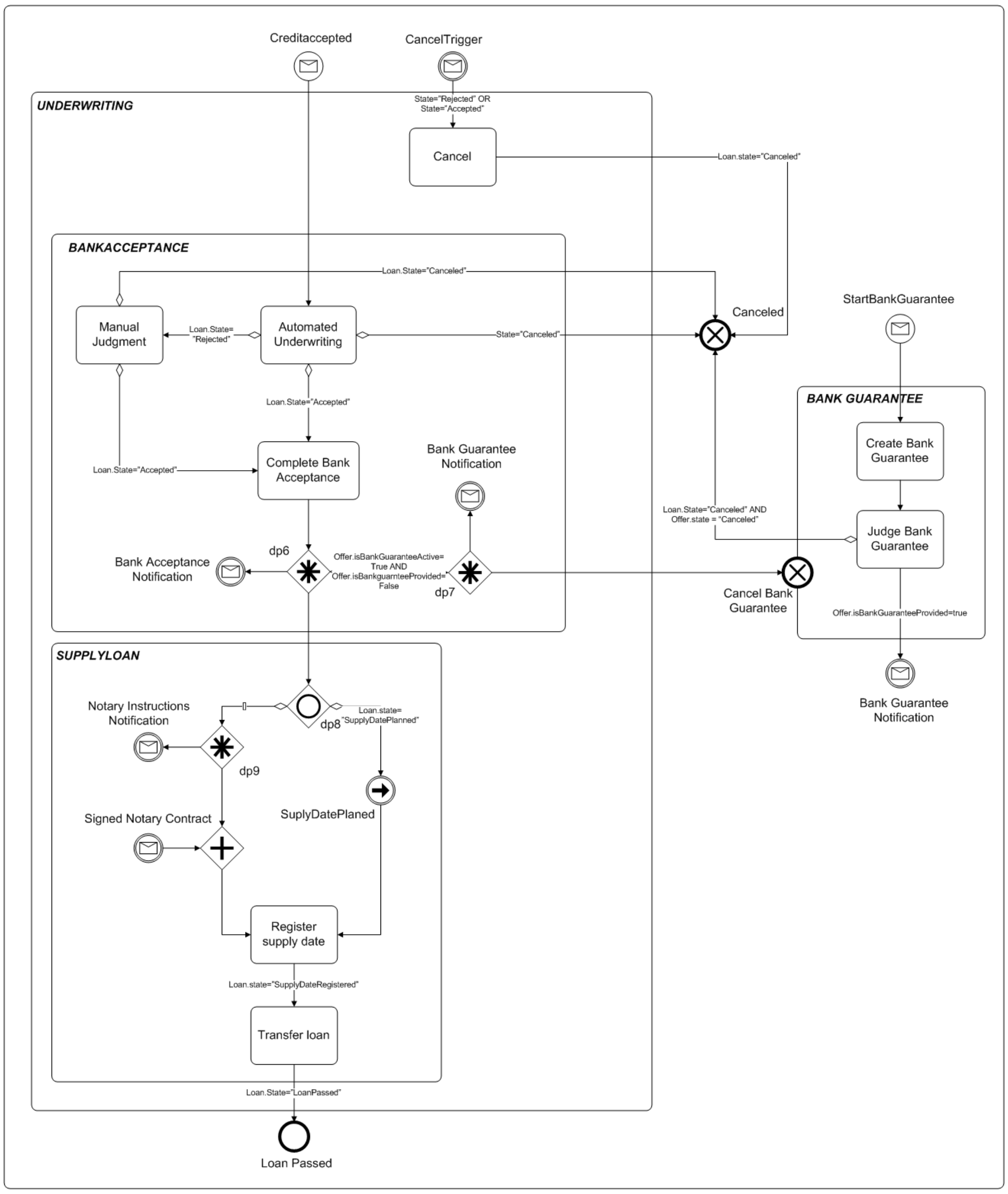


FIGURE D.6: Underwriting sub-process example.

- $C_3$, is translated into a maximal component composed of a sequence component and the end point event of the sub-process application. Notice that $C_3$ is identified as an independent component, considering a maximal component composed of $C_2$ and $C_3$ would introduce an arbitrary cycle (from line 96 to 104 in Listing D.1).
- $C_4$, is translated into a cancel region, where the sub-process application is canceled thorough a trigger event if the specified condition is satisfied (from line 11 to 25 in Listing D.1).

### STEP4: Translate the sub-process application into BPEL code

Following the fourth step of our transformation approach we can produce the BPEL code for the sub-process application (see Listing D.1). Notice that $C_1$ (at line 107 in Listing D.1) is the start point of the sub-process and in $C_3$ (at line 108 in Listing D.1) triggers the end of the sub-process. Additional the cancel trigger introduces a fault handler for the corresponding scope activity for the sub-process application. The fault handler should perform the task cancel before the the scope is exited. The fault handler *fCANCELTRIGGER* is introduced (from line 2 to 9 in Listing D.1)to remove all process instances before the task cancel is performed.

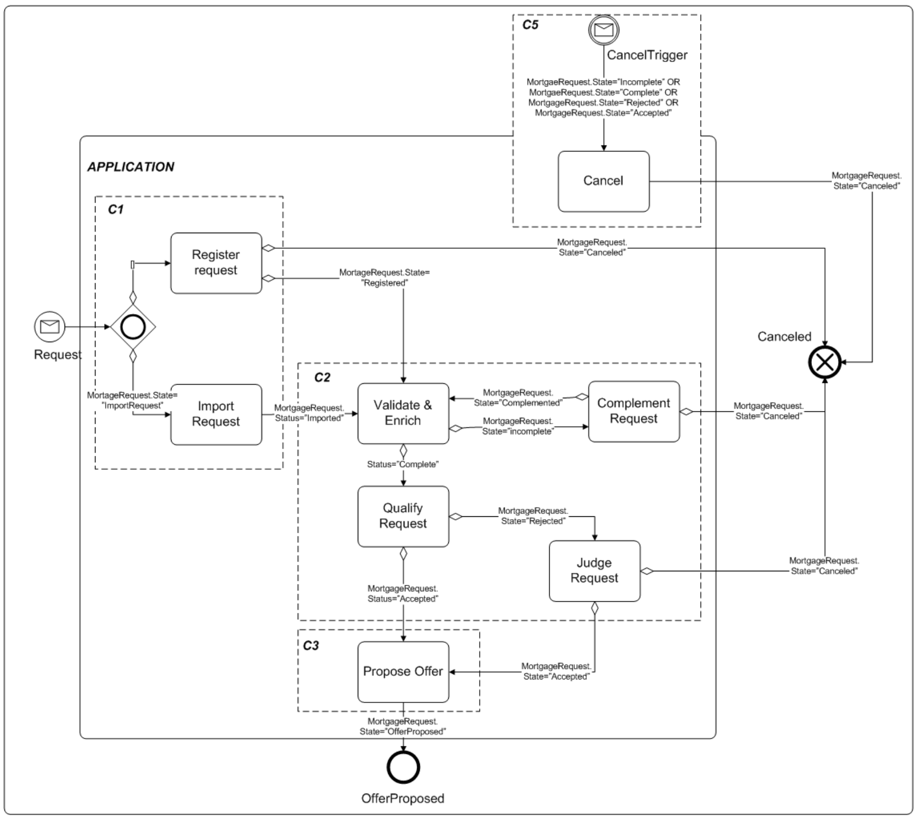


FIGURE D.7: Components in the sub-process application.

```
1  <scope name="APPLICATION">
2   <faultHandler>
3    <catch faultName="fCANCELTRIGGER">
4     <sequence>
5      <invoke operation="Cancel"/>
6      <invoke operation="Start(Canceled)"/>
7     </sequence>
8    <catch>
9   </faultHandler>
10  <eventHandler>
11   <!--------------- Event Action translation of C4 --------------------->
12   <onMessage operation="Start(CANCELTRIGGER)">
13    <scope name="C4">
14     <switch>
```

```
    <case condition="MortageRequest.state='Incomplete' OR
     MortgageRequest.state='Complete' OR MortgageRequest.state='Rejected'
     OR MortgageRequest.state='Accepted'">
      <throw faultName="fCANCELTRIGGER"/>
    </case>
    <otherwise>
     </empty>
    </otherwise>
   </switch>
  </scope>
 </onMessage>
 <!--------------- Event Action translation of C1 --------------------->
 <onMessage operation="Start(C1)">
  <scope name="C1">
   <switch>
    <case condition="MortgageRequest.state="importRequest">
     <sequence>
      <invoke operation="Import_request"/>
      <invoke operation="Start(C2)"/>
     </sequence>
    </case>
    <otherwise>
     <sequence>
      <invoke operation="Register_request"/>
      <switch>
       <case condition='MortgageRequest.state="registered'">
        <invoke operation="Start(C2)"/>
       </case>
       <otherwise>
        <invoke operation="Start(Canceled)"/>
       </otherwise>
      </switch>
     </sequence>
   </switch>
  </scope>
 </onMessage>
 <!--------------- Event Action translation of C2 --------------------->
 <onMessage operation="Start(C2)">
  <scope name="C2">
   <sequence>
    <invoke operation="Validate_and_enrich"/>
    <switch>
    <case condition="MortgaageRequest.state='Complete'">
     <sequence>
      <invoke operation="Qualify_request"/>
      <switch>
       <case condition="Accepted">
        <invoke operation="Start(C3)"/>
       </case>
       <otherwise>
        <sequence>
         <invoke operation="Judge_request"/>
         <switch>
          <case condition="Accepted">
           <invoke operation="Start(C3)"/>
           </case>
           <otherwise>
            <invoke operation="Start(Canceled)"/>
           </otherwise>
          </switch>
         </sequence>
        </otherwise>
       </switch>
      </case>
      <otherwise>
       <sequence>
        <invoke operation="Complement_request"/>
        <switch>
         <case condition="Complemented">
          <invoke operation="Start(C2)"/>
         </case>
         <otherwise>
          <invoke operation="Start(Canceled)"/>
         </otherwise>
        </switch>
       </otherwise>
      </sequence>
     </switch>
    </sequence>
   </scope>
  </onMessage>
  <!--------------- Event Action translation of C3 --------------------->
  <onMessage operation="Start(C3)">
   <scope name="C3">
    <sequence>
     <invoke operation="Propose_offer"/>
     <invoke operation="End(C3)">
    </sequence>
   </scope>
  </onMessage>
 <!--------------- Start point of the scope application ----------------->
 <sequence>
  <invoke operation="Start(C1)"/>
  <receive operation="End(C3)"/>
  <invoke operation="Start(OfferProposed)"/>
 </sequence>
</scope>
```

Listing D.1: Translation of the the sub-process application

### D.2.2 Transform the the sub-process processing into BPEL

**STEP3: Decomposed the sub-process offerprocessing into components**

Following the second step of our transformation approach we can identify the following components for the sub-process offerprocessing (see Figure D.8):

- $C_1$, is translated into a maximal component composed of a deferred choice and exclusive choice activating either $C_2$, $C_3$, $C_5$ or $C_{10}$ (from line 52 to 77 in Listing D.2).
- $C_2$, cannot be translated into a BPEL because is consist a typical WP9. This is caused as when the first document is inserted it should be possible that both the task update task is activated and in parallel it is still allowed to insert new documents until after the last document the file is considered to be complete. (from line 78 to 91 in Listing D.2).
- $C_3$, is translated into an maximal component composed of exclusive choice resulting either in cancellation or the activation of $C_4$ (from line 92 to 107 in Listing D.2). Notice that $C_3$ is identified as an independent component, considering a maximal component composed of $C_3$ and $C_4$ would introduce an arbitrary cycle.
- $C_4$, is translated into a sub-set of multiple exclusive choices in sequence with a multi choice. The completion of $C_4$ results either in completion of the sub-process offerprocessing, cancellation of the process or activation of $C_4$ (from line 108 to 149 in Listing D.2).
- $C_5$, is translated into maximal component composed of a sequence of a task and results in the cancellation of the process (from line 150 to 159 in Listing D.2).
- $C_6$, is translated into a cancel region component, where the event trigger activates the cancellation of the process in $C_5$ (from line 26 to 39 in Listing D.2).
- $C_7$, is translated into a cancel region component, where a timer exception *Offer-Expired* creates an instance of $C_7$ (from line 40 to 45 in Listing D.2).
- $C_8$, is translated into a cancel region component, where a timer exception *OfferAlmostExpired* creates an instance of $C_7$ (from line 46 to 51 in Listing D.2).

**STEP4: Translate the sub-process offerprocessing into BPEL**

Following the third step of our transformation approach we can produce the BPEL code for the sub-process offerprocessing as presented in listing D.2). Notice that component $C_1$ (at line 162 in Listing D.2) is the start point and the component $C_4$ (at line 163 Listing D.2) the end of the sub-process application. Aditionally, the cancel trigger should remove all instances of the sub-process offerprocessing and performs the component $C_5$ before before the sub-process application is canceled. Therefore an faulthandler *fCANCELTRIGGER* is introduced to remove all process instances before the component $C_5$ is performed. Notice that due to the pattern WP9 it is not possible to translate the sub-process process completely into BPEL code.

**STEP3: Decompose the sub-process processing into components**

Following the thrid step of our transformation approach we can identify the following component for the sub-process processing:

- $C_9$, the sub-process offer processing is contained by the sub-process processing, such that the translated BPEL code for this sub-process is nested into the scope activity of the sub-process processing (at line 163 in Listing D.2).
- $C_{10}$, is translated into a sequence of an task and a multi-choice resulting in either cancellation or continuation of the sub-process processing (from 3 to 14 in Listing D.2).

**STEP4: Translate the sub-process processing into BPEL**

Following the fourth step of our transformation approach we can produce the BPEL code for the sub-process processing as presented in listing D.2). Notice that the resulting process after the third step does not contain an arbitrary cycle or unstructured components or additional fault handlers, such that in this case the fourth step is not required.

```
1  <scope name="PROCESSING">
2   <eventHandlers>
3   <!--------------- Event Action translation of C10 -------------------->
4    <onMessage operation="Start(C10)">
5     <scope name="C10">
6      <sequence>
7       <invoke operation="Extend_sign_period"/>
8       <flow suppressJoinFailure="yes">
9        <invoke operation="Recall_notifcation"/>
10       <invoke operation="Start(Canceled)"
11        transitionCondition="Offer.state=Canceled"/>
12      </flow>
13     </sequence>
14    </scope>
15   </onMessagge>
16  </eventHandlers>
17  <!--------------- Start point of the scope processing  ----------------->
18  <scope name="C9">
19   <scope name="OFFERPROCESSING">
20    <faultHandler>
21     <catch faultName="fCANCELTRIGGER">
22      <invoke operation="Start(C5)"/>
23     </catch>
24    </faultHandler>
25    <eventHandler>
26     <!--------------- Event Action translation of C6 -------------------->
27     <onMessage operation="Start(CANCELTRIGGER)">
28      <scope name="C6">
29      <switch>
30       <case condition="Offer.state='DocumentsInComplete' OR
```

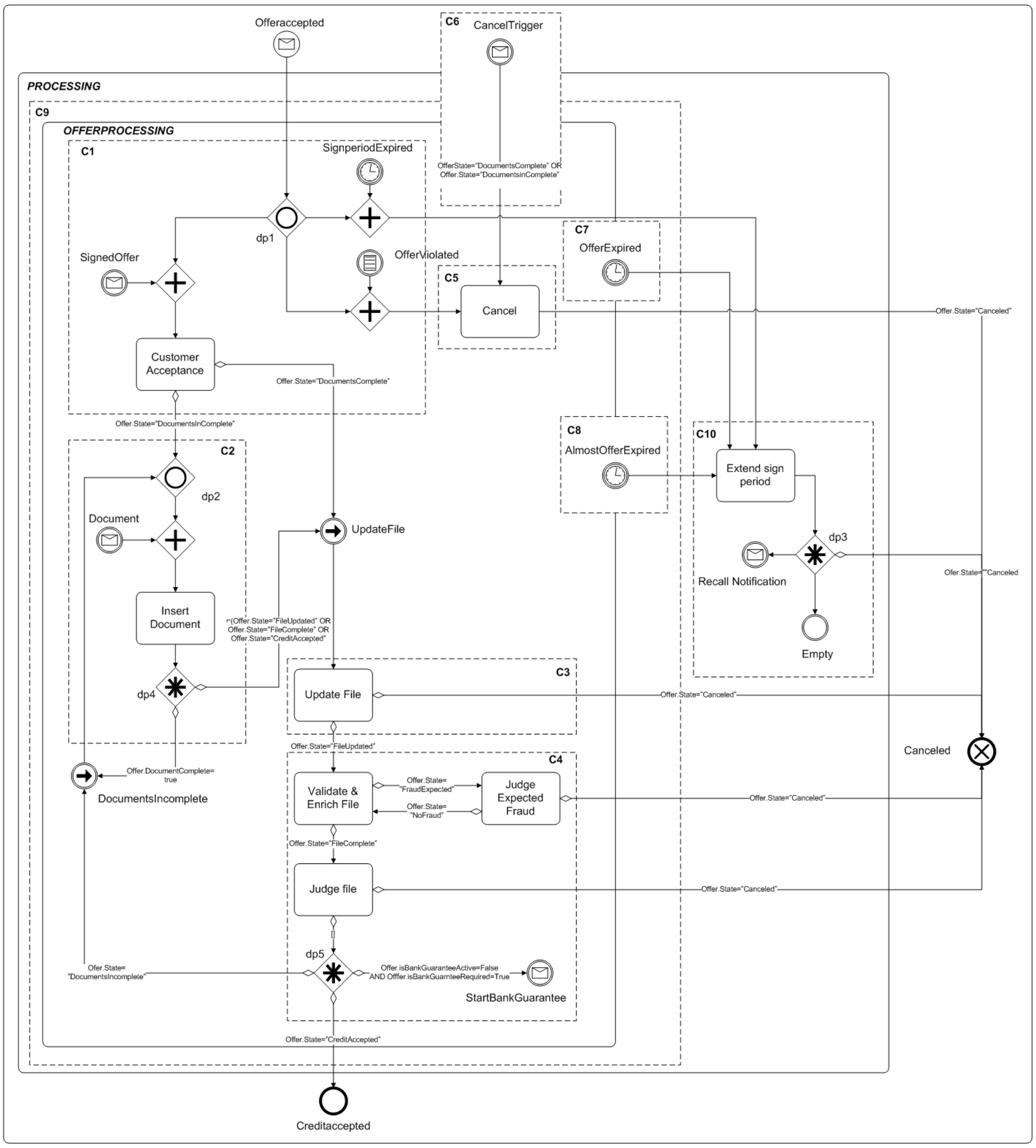


FIGURE D.8: Components in the sub-process processing.

```
       Offer.state='DocumentsComplete'">
        <throw faultName="fCANCELTRIGGER"/>
       </case>
       <otherwise>
        </empty>
       </otherwise>
      </switch>
     </scope>
    </onMessage>
    <!--------------- Event Action translation of C7 -------------------->
    <onAlarm name="OfferExpired">
     <scope name="C7">
      <invoke operation="Start(C10)"/>
     </scope>
    </onAlarm>
    <!--------------- Event Action translation of C8 -------------------->
    <onAlarm name="AlmostOfferExpired">
     <scope name="C8">
      <invoke operation="Start(C10)"/>
     </scope>
    </onAlarm>
    <!--------------- Event Action translation of C1 ------------------->
    <onMessage operation="Start(C1)">
     <scope name="C1">
      <pick>
       <onMessage operation="SignedOffer">
        <sequence>
         <invoke operation="Customer_acceptance">
         <switch>
          <case condition="Offer.state='DocumentsInComplete'">
           <invoke operation="Start(C2)"/>
          </case>
          <otherwise>
           <invoke operation="Start(C3)"/>
          </otherwise>
         </switch>
        </sequence>
       </onMessage>
       <onAlarm name="SignedPeriodExpired">
        <invoke operation="Start(C5)"/>
       </onAlarm>
       <onMessage operation="OfferViolated">
        <invoke operation="Start(Canceled)"/>
       </onMessage>
      </pick>
     </scope>
    </onMessage>
    <!--------------- Event Action translation of C2 ------------------->
    <onMessage operation="Start(C2)">
     <scope name="C2">
      <-- START NEED TO BE RESOLVED BECAUSE OF WP9 -->
      ?-----------------------------------------------
      <sequence>
        <receive operation="Document"/>
        <invoke operation="Insert_document"/>
       <invoke operation="Start(C4)"/>
      <sequence>
      ?--------------------------------------------------
      <-- END NEED TO BE RESOLVED BECAUSE OF WP9 -->
     </scope>
    </onMessage>
    <!--------------- Event Action translation of C3 -------------------->
    <onMessage operation="Start(C3)">
     <scope name="C3">
      <sequence>
       <invoke operation="Update_File"/>
       <switch>
        <case condition="Offer.state='FileUpdated'">
        <invoke operation="Start(C4)"/>
        </case>
        <otherwise>
         <invoke operation="Start(Canceled)"/>
        </otherwise>
       </switch>
      </sequence>
     </scope>
    </onMessage>
    <!--------------- Event Action translation of C4 --------------------->
    <onMessage operation="Start(C4)">
    <scope name="C4">
     <sequence>
      <invoke operation="Validate_enrich">
      <switch>
       <case condtion="Offer.state="FileComplete">
        <invoke operation="Judge_file"/>
        <switch>
         <case condition="State='Canceled'">
          <invoke operation="Start(Canceled)"/>
         </case>
         <otherwise>
          <flow suppressJoinFailure>
           <invoke operation="Start(BANKGUARANTEE)"
            joinCondition="Offer.BankGuaranteeActive=False AND
            Offer.BankGuaranteeRequered=True"/>
           <invoke operation="Start(C2)"
            joinCondition="Offer.state='DocumentsIsComplete'"/>
           <invoke operation="End(C4)"
            joinCondition="state='CreditAccepted'"/>
          </flow>
         </otherwise>
        </switch>
       </case>
       <otherwise>
        <sequence>
         <invoke operation="Judge_expected_fraud"/>
         <switch>
           <case condition="Offer.state='NoFraud'">
            <invoke operation="Start(C4)"/>
          </case>
          <otherwise>
```

```
          <invoke operation="Start(Canceled)"/>
         </otherwise>
        </switch>
       </sequence>
      </otherwise>
     </switch>
    </sequence>
   </scope>
  </onMessage>
  <!--------------- Event Action translation of C5 --------------------->
  <onMessage operation="Start(C5)">
   <scope name ="C5">
    <sequence>
     <invoke operation="Cancel">
     <invoke operation="Start(Canceled)"/>
    </sequence>
   </scope>
  </onMessage>
 </eventHandler>
 <!--------------- Start point of the scope processing  ---------------->
 <sequence>
  <invoke operation="Start(C1)">
  <receive operation="End(C4)"/>
  <invoke operation="Credit_accepted"/>
 </sequence>
</scope>
</scope>
</scope>
```

LISTING D.2: Translation of the sub-process processing

### D.2.3 Transform the the sub-process underwriting and bankguarantee into BPEL

**STEP3: Decompose the sub-process bankacceptance into components**

Following the third step of our transformation approach we can identify the following components for the sub-process bankaccepatance (see Figure D.9):

- $C_1$, is translated into a maximal component composed of exclusive choices, which result either in cancellation or the activation of $C_2$ (from line 30 to 56 in Listing D.3).
- $C_2$, is translated into a maximal component composed of sequence of a task and an multi choices, resulting in the activation of the sub-process supply loan and (only if the governing condition is satisfied) the cancellation of the sub-process bankguarantee (from line 57 to 78 in Listing D.3).

**STEP4: Translate the sub-process bankaccepance into BPEL**
Following the fourth step of our transformation approach we can produce the BPEL code for the sub-process application as presented in listing D.2). The component $C_1$ is the start point (at line 81 in Listing D.3) and (at line 82 in Listing D.3) waits for the end of the component $C_2$ in sub-process bankacceptance. Notice that no additional fault handlers are required for the sub-process bankacceptance.

**STEP3: Decompose the sub-process supplyloan into components**
Following the third step of our transformation approach we can identify the following components for the sub-process supplyloan:

- $C_3$, is translated into a maximal component composed of a one way, sequence and exclusive choice (from line 86 to 106 in Listing D.3).

**STEP4: Translate the sub-process supplyloan into BPEL**
Following the third step of our transformation approach we can produce the BPEL code for the sub-process supplyloan as presented in listing D.2). Notice that the resulting process after the second step does not contain an arbitrary cycle or unstructured components, such that third step is not required. Notice that no additional fault handlers are required for the sub-process supplyloan.

**STEP3: Decompose the sub-process underwriting components**
Following the third step of our transformation approach we can identify the following components for the sub-process underwriting:

- $C_4$, is translated into a maximal component composed of a sequence of the sub-process bankacceptance and supplyloan (from line 25 to 106 in Listing D.3).
- $C_5$, is translated into a cancel region component, where the sub-process application is canceled through a trigger event if the specified condition is satisfied. Cancellation is performed explicitly in after the cancel task, such that no additional fault handler is required in the scope of the application (from line 11 to 23 in Listing D.3).

**STEP4: Translate the sub-process underwriting into BPEL**
The sub-process underwriting does not contain an arbitrary cycle or unstructured components after the second step, such that the the third sept is not required. The cancel trigger should remove all instances of the sub-process unerwriging and performs the cancel task before before the sub-process application is canceled. Therefore an faulthandler *fCANCELTRIGGER* is introduced to remove all process instances before the cancel task is performed.

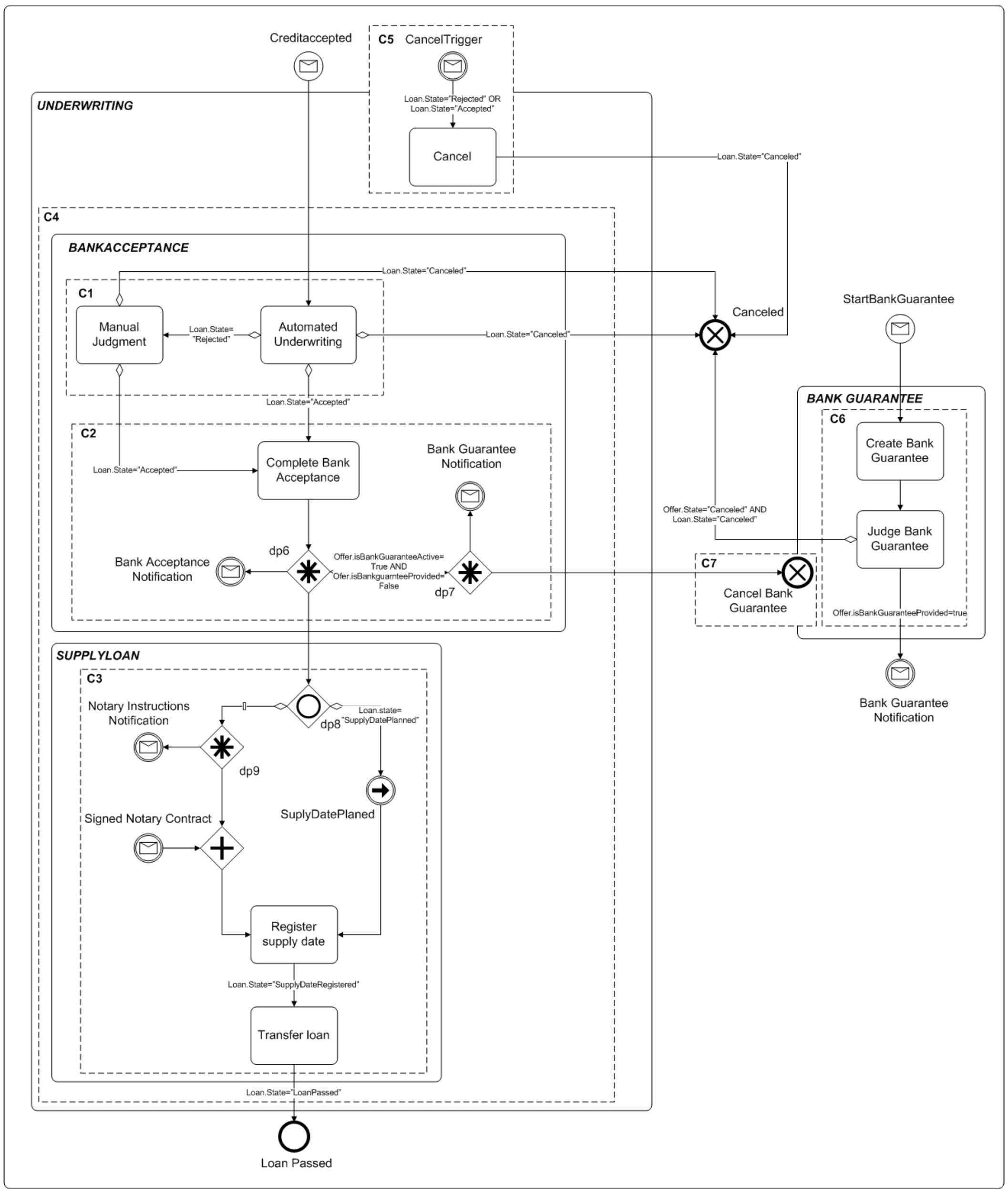


Figure D.9: Components in the sub-process underwriting and bankguarantee.

```
<scope name="UNDERWRITING">
 <faultHandler>
        <catch faultName="fCANCELTRIGGER">
         <sequence>
    <invoke operation="Cancel"/>
    <invoke operation="Start(Canceled)"/>
   </sequence>
        </catch>
 </faultHandler>
 <eventHandler>
  <!--------------- Event Action translation of C5 ----------------------->
  <onMessage operation="Start(CANCELTRIGGER)">
   <scope name="C5">
   <switch>
    <case condition="Loan.state='Rejected' OR Loan.state='Accepted'">
     <throw faultName="fCANCELTRIGGER"/>
    </case>
    <otherwise>
     </empty>
    </otherwise>
   </switch>
  </scope>
 </onMessage>
</eventHandler>
<!--------------- Translation of C4 -------------------------------------->
<scope name="C4">
 <sequence>
  <scope name="BANKACCEPTANCE">
   <eventHandler>
    <!--------------- Event Action translation of C1 --------------------->
    <onMessage operation="Start(C1)">
     <sequence>
      <invoke operation="Automated_underwriting"/>
      <switch>
       <case condition="Loan.state='Accepted'">
        <invoke operation="Start(C3)"/>
       </case>
       <case condition="Loan.state= 'Rejected'">
        <sequence>
         <invoke operation="Manual_Judgement"/>
         <switch>
          <case condition="Loan.state='Accepted'">
           <invoke operation="Start(C3)"/>
          </case>
          <otherwise>
           <invoke operation="Start(Canceled)"/>
          </otherwise>
         </switch>
        </sequence>
       </case>
       <otherwise>
        <invoke operation="Start(Canceled)"/>
       </otherwise>
      </switch>
     </sequence>
    </onMessage>
    <!--------------- Event Action translation of C2 --------------------->
    <onMessage operation="Start(C2)">
    <scope name="C2">
     <sequence>
      <invoke operation="Complete_Bankacceptance"/>
      <sequence>
       <invoke operation="Complete_BankAcceptance"/>
        <flow suppressJoinFailure="yes">
        <invoke operation="BankAccepatance_notificaiton"/>
        <invoke operation="Start(CANCELBANKGUARANTEE)"
         joinCondition="Offer.BankGuarnteeActive=true AND
         Offer.BankGuarnteeProvided=false"/>
         <invoke operation="BankGuarantee_notification"
          joinCondition="Offer.BankGuarnteeActive=true AND
          Offer.BankGuarnteeProvided=false"/>
        </flow>
       </sequence>
       <invoke operation="End(C2)"/>
      </sequence>
     </scope>
    </onMessage>
   </eventHandler>
   <!--------------- Start point of the scope bankacceptance  ----------->
   <sequence>
     <invoke operation="Start(C1)"/>
     <receive operation="End(C2)">
   </sequence>
  </scope>
  <scope name="SUPPLYLOAN">
   <!--------------- Event Action translation of C3 --------------------->
   <scope name="C3">
    <sequence>
     <switch>
      <case condition="Loan.state='SupplyDatePlanned'">
       <empty/>
      </case>
      <otherwise>
       <sequence>
        <invoke operation="NotaryInstructions_Notification"/>
        <receive operation="Signed_Notarty_Contract"/>
       </sequence>
      </otherwise>
     </switch>
    <sequence>
     <invoke operation="Register_Supply_Date"/>
     <invoke operation="Transfer_Loan"/>
    </sequence>
   </scope>
  </sequence>
 </scope>
 <invoke operation="Loan_passed"/>
</scope>
```

Listing D.3: Translation of the sub-process underwriting

**STEP3: Decompose the sub-process bankguarnatee components into BPEL** Following the third step of our transformation approach we can identify the following components for the sub-process bankgurantee (see Figure D.9):

- $C_6$, is translated into a maximal component composed of a sequence and exclusive choice, resulting in either the completion of the sub-process bankguarntee or cancellation of the process (from line 18 to 32 in Listing D.4).

- $C_7$, is translated into a cancel task component, for which the occurrence of a cancelbankguarntee event results in the termination of the all active instances in the sub-process bankguarantee (from line 11 to 16 in Listing D.4).

**STEP4: Translate the sub-process bankgurantee into BPEL**

The sub-process bankguarntee does not contain an arbitrary cycle or unstructured components after the third step, but the sub-process bankguarantee contains a cancel sub-process event. This requires an aditional faulthandler in the scope activity resembeling the sub-process bankguarantee (from line 2 to 7 in Listing D.4).

```
 1 <scope name="BANKGUARANTEE">
 2  <!---------------Fault handler required for the cancel task of C7----------->
 3  <faultHandler>
 4   <catch faultName="fCANCELBANKGUARANTEE">
 5    </empty>
 6   </catch>
 7  </faultHandler>
 8  <eventHandler>
 9   <!--------------- Event hander required for the cancel task---------------->
10   <onMessage operation="Start(CANCELBANKGUARANTEE)">
11    <scope name="C7">
12     <sequence>
13      <throw faultName="fCANCELBANKGUARANTEE"/>
14     </sequence>
15    </scope>
16   <onMessage>
17  </eventHandler>
18  <!--------------- Event Action translation of C6 --------------------------->
19  <scope name="C6">
20   <sequence>
21    <invoke operation="Create_BankGuarantee"/>
22    <invoke operation="JudgeBankGuarantee"/>
23    <switch>
24     <case condition="BankGuaranteeProvide">
25      <invoke operation="BankGuarntee_notification"/>
26     </case>
27     <otherwise>
28      <invoke operation="Start(Canceled)"/>
29     </otherwise>
30    </switch>
31   </sequence>
32  </scope>
33 </scope>
```

Listing D.4: Translation of the sub-process bankguarantee

### D.2.4 Transform the the mortgage process into BPEL

**STEP3: Decompose the mortgage process into components**

Following the third step of our transformation approach we can identify the following components for the mortgage process (see Figure D.10):

- $C_1$, is translated into a multiple start point event element, resulting in the activation of $C_2$ (from line 15 to 27 in Listing D.5).
- $C_2$, is translated into a single sub-process application, resulting in the activation of $C_3$ (from line 28 to 34 in Listing D.5).
- $C_3$, is translated into a maximal component composed of sequence of the sub-processes processing and underwriting resulting in the completion of the process (from line 35 to 42 in Listing D.5).
- $C_4$, is translated into a single sub-process bankgurantee, for which the completion depends on the cancel task pattern as implemented in the underwriting (from line 43 to 46 in Listing D.5).
- $C_5$, is translated into a maximal sequence composed of a deferred choice and exclusive choice resulting in the termination of the process or the activation of either $C_2$ or $C_3$ (from line 47 to 67 in Listing D.5).

**STEP4: Translate the mortgage process into BPEL**
Following the fourth step of our transformation approach we can produce the BPEL code for the sub-process application (see Listing D.5). Notice that $C_1$ is the start point (at line 70 in Listing D.5) and waits for the completion of $C_3$ (at line 71 in Listing D.5) triggers the end of the mortgage process. The mortgage process is responsible to handle the cancellation, such that an event handler *Start(Canceled)* is introduced to handle all cancel events from the underlying sub-proceses. This cancel event handler throws an fault, such that all running instances in the process are removed, such that a single instance continues processing $C_5$.

```
 1 <process name="MORTGAGEPROCESS">
 2 <faultHandler>
 3    <!-- Remove all instances of the process and process continuation -->
 4    <catch faultName="fCanceled">
 5     <invoke operation="Start(C5)">
 6    </catch>
 7 </faultHandler>
 8
 9 <eventHandler>
10  <!--------------- Translation of the Cancelation --------------------->
11   <onMessage operaton="Start(Canceled)">
12    <throw faultName="fCanceled"/>
13   </onMessage>
14
15  <!--------------- Event Action translation of C1 --------------------->
16  <onMessage operation="Start(C1)">
17   <scope name="C1">
```

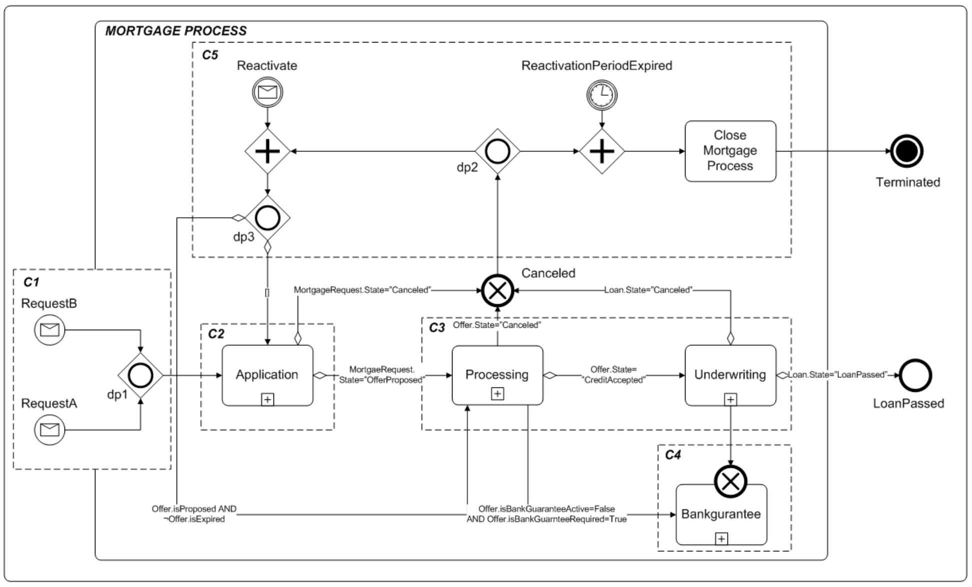

MORTGAGE PROCESS
C5
Reactivate
ReactivationPeriodExpired
Close Mortgage Process
dp2
Terminated
dp3
C1
RequestB
RequestA
dp1
C2
Application
MortgageRequest.State="Canceled"
Canceled
Loan.State="Canceled"
C3
Offer.State="Canceled"
MortgaeRequest. State="OfferProposed"
Processing
Offer.State= "CreditAccepted"
Underwriting
Loan.State="LoanPassed"
LoanPassed
C4
Bankgurantee
Offer.isProposed AND ¬Offer.isExpired
Offer.isBankGuaranteeActive=False AND Offer.isBankGuarnteeRequired=True

FIGURE D.10: Components in the mortgage process.

```
    <pick>
     <onMessage operation="RequestA" createInstance="yes">
      <invoke operation="Start(C2)"/>
     </onMessage>
     <onMessage operation="RequestB" createInstance="yes">
      <invoke operation="Start(C2)"/>
     </onMessage>
    </pick>
   </scope>
  </onMessage>
  <!--------------- Event Action translation of C2 --------------------->
   <onMessage operation="Start(C2)">
    <sequence>
      <!-- code for sub-process APPLICATION -->
     <invoke operaton="Start(C3)"/>
    </sequence>
   </onMessage>
  <!--------------- Event Action translation of C3 --------------------->
   <onMessage operation="Start(C3)">
    <sequence>
     <!-- code for sub-process PROCESSING -->
     <!-- code for sub-process UNDERWRITING -->
     <invoke operation ="End(C3)">
    </sequence>
   </onMessage>
  <!--------------- Event Action translation of C4 --------------------->
  <onMessage operation="Start(C4)">
   <!-- code for sub-process BANKGUARANTEE -->
  </onMessage>
  <!--------------- Event Action translation of C5 --------------------->
  <onMessage operation="Start(C5)">
   <pick>
    <onMessage operation="Reactivate">
     <switch>
```

```
      <case condition="Offer.isProposed AND NOT Offer.isExpired">
       <invoke operation="Start(C2)"/>
      </case>
      <otherwise>
       <invoke operation="Start(C2)">
      </otherwise>
     </switch>
    </onMessage>
    <onAlarm name="ReactivationPeriodExpired">
     <sequence>
      <invoke operation="Close Mortgage Process"/>
      <terminate name="terminated"/>
     </sequence>
    </onAlarm>
   </pick>
  </onMessage>
  <!--------------- Start point of the scope mortgageprocess  -- ------->
  <sequence>
   <invoke operation="Start(C1)"/>
   <receive operation="End(C3)"/>
   <invoke operation="LoanPassed"/>
  </sequence>
  </scope>
</process>
```

Listing D.5: Translation of the the mortgage process