UNIVERSITY OF BELGRADE
FACULTY OF ORGANIZATIONAL SCIENCES

MILAN V. MILANOVIĆ

MODELING RULE-DRIVEN SERVICE ORIENTED ARCHITECTURES

PHD THESIS

Belgrade, 2010.

SUPERVISED BY:

Dr Vladan Devedžić, Professor
Faculty of Organizational Sciences, University of Belgrade, Serbia

COMITTEE MEMBERS:

Dr Dragan Djurić, Assistant Professor
Faculty of Organizational Sciences, University of Belgrade, Serbia

Dr Siniša Vlajić, Assistant Professor
Faculty of Organizational Sciences, University of Belgrade, Serbia

Dr Dragan Gašević, Associate Professor
School of Computing and Information Systems, Athabasca University, Edmonton, Canada

Dr Dragan Bojić, Assistant Professor
School of Electrical Engineering, University of Belgrade, Serbia

Date of PhD thesis defence: ________________

Date of PhD thesis promotion: ________________

Modelovanje servisno orijentisanih arhitektura korišćenjem pravila

Apstrakt:

Ova doktorska disertacija je fokusirana na dizajniranje i implementacjiu jezika za modelovanje poslovnih procesa upotrebom pravila za servisno orijentisane arhitekture (SOA). Jezik je baziran na de facto standardu za modelovanje poslovnih procesa (tj. BPMN-u) i generalnom jeziku za definisanje pravila (R2ML-u). Postojeća rešenja u ovoj oblasti su pokazala da su procesno orijentisani modeli suviše nefleksibilna za dinamičku adaptaciju poslovne logike. Rešenja bazirana na pravilima daju alternativu, koja nudi veću fleksibilnost zahvaljujući deklarativnoj prirodi pravila i njihovim algoritmima rezonovanja. Međutim, modelovanje poslovnih procesa korišćenjem pravila je zamoran proces za programere u odnosu na ukupno razumevanja poslovnih procesa.

U ovoj disertaciji je predložen hibridni pristup, gde jezik za modelovanje uključuje procesno orijentisanu, ali i na pravilima baziranu perspektivu. Jezik (na pravilima bazirani BPMN – rBPMN) se koristi za modelovanje različitih tipova kompozicija servisno orijentisanih arhitektura, kao što su orkestracije i koreografije. Prethodna istraživanje u domenu modelovanja orkestracija su pokazala da: i) dobre prakse za dijagrame toka nisu najbolje pokrivene u postojećim jezicima, ii) jezici za modelovanje poslovnih procesa imaju ograničenu podršku za reprezentaciju logičkih izraza i pravila, iii) ograničena je podrška za dinamičke promene delova poslovne logike u izvršnim orkestracijama servisa, i iv) postoji potreba za metodologijom, koja bi dozvolila sistemsku upotrebu tri ključna aspekta koja doprinose modelovanju orkestracija servisa – rečnici, pravila i procesi. Kako bi odgovorili na ove zahteve, u ovom radu se prelaže metodologija za sistematsko definisanje koraka za proces razvoja servisno orijentisanih arhitektura.

Istraživačka zajednica je bila uglavnom fokusirana na problem modelovanja orkestracija servisa u domenu kompozicija servisa, dok je modelovanje koreografija zauzimalo manje mesta u tim istraživanjima. Sledeći zahtevi u domenu modelovanja koreografija su analizirani u ovoj disertaciji: i) modeli koreografija nisu dobro spojeni sa rečnicima modela, ii) ograničena je podrška za razdvajanje delova poslovne logike od modela koreografija. Ovo smanje mogućnost dinamičkih promena u koreografijama, iii) modeli koreografija sadrže suvišne elemente deljene poslovne logike, što može voditi ka nekonzistetnosti implementacije i nekompatibilnom ponašanju.

Kako bi evaluirali rBPMN jezik u odnosu na različite vrste kompozicija servisa i kako bi uporedili dato rešenje sa postojećim rešenjima, dali smo pregled uzora kod razmene poruke, uzora za interakciju servisa i uzora za kontrolu toka kod modela orkestracije, kao i agilnih uzora kako bi evaluirali dinamičnost našeg rešenja. Takođe smo pokazali kako se razvijeni jezik može koristiti u različitim studijama slučajeva korišćenja za modelovanje realnih poslovnih procesa.

Takođe smo razvili softversko okruženje bazirano na Eclipse platformi, pod nazivom rBPMN editor, koje uključuje implementaciju rBPMN jezika, kao i grafički editor za definisanje na pravilima baziranim poslovnih prosela u rBPMN jeziku. Pored opisa dizajna i implementacije razvijenog softverskog rešenja, ova disertacija pruža i komparativnu analizu rBPMN jezika sa drugim jezicima u oblasti modelovanja poslovnih procesa.

Ključne reči:

Poslovni procesi, poslovna pravila, Meta-modeli, BPMN, R2ML, rBPMN, metodologija

Modeling rule-driven Service Oriented Architectures

Abstract:

This PhD thesis is focused on the design and implementation of a novel rule-based business process language for modeling Service Oriented Architectures (SOA). The proposed language is built on a de facto standard for process modeling (i.e., BPMN) and a general rule markup language (R2ML). The existing solutions to this topic demonstrated that process-oriented models might be too rigid for dynamic adaptations of the business logic. Rule-based approaches are considered an alternative, which offers more flexibility thanks to the declarative nature of rules and their underlying reasoning algorithms. However, modeling a business process through rules is a tedious process for developers in terms of the overall business process comprehension.

In this thesis, we propose a modeling language that integrates both rule- and process-oriented modeling perspectives of a business process. The language (rule-based BPMN – rBPMN) is used to model different types of SOA compositions, including orchestrations and choreographies. Regarding the orchestrations, the previous research on business process modeling of service orchestrations, demonstrated that: i) best practices for workflows are not fully covered in the existing languages; ii) business process languages have limited support for representing logical expressions and rules; iii) there is a limited support for dynamic changes of parts of business logic in executable service orchestrations; and iv) there is a need for a methodology, which allows for systematic use of the three key aspects contributing to the modeling of service orchestrations – business vocabularies, rules, and processes. In order to address these challenges, in addition to the rBPMN language, in this thesis, we propose a methodology for defining a systematic set of steps for the development process of service oriented architectures.

The research community has so far mainly focused on the problem of modeling of service orchestrations in the domain of service composition, while modeling of service choreographies has attracted less attention. The following identified challenges in choreography modeling are tackled in this thesis: *i)* choreography models are not well-connected with the underlying business vocabulary models. *ii)* there is limited support for decoupling parts of business logic from complete choreography models. This reduces dynamic changes of choreographies; *iii)* choreography models contain redundant elements of shared business logic, which might lead to an inconsistent implementation and incompatible behavior.

In order to evaluate the rBPMN language for different service compositions and to compare our approach with related solutions, we leverage message exchange patterns, service-interaction patterns for choreography models, control flow patterns for orchestration models, and agility patterns for evaluation of dynamicity of business processes. In addition, we show how the developed language can be used in different case studies to model real world business processes.

To have a proof of concept, we developed a software environment based on Eclipse, called rBPMN Editor, which includes implementation of the rBPMN language and also a graphical editor for defining rule-based business processes in the rBPMN language. Along with the description of the design and implementation of the developed software environment, the thesis provides a comparative analysis of the rBPMN language with other similar languages in the area of modeling business processes.

Keywords:

Business processes, Business rules, Meta-models, Model-Driven Engineering, BPMN, R2ML, rBPMN, methodology

CONTENTS

1. Introduction

Service-oriented architecture (SOA) is a software paradigm for building flexible and loosely coupled software systems based on services. Services are software entities that can be easily discovered, published and described. SOA approach to creation of software sysetms enables assembling applications independent of specific platform by discovering and calling services to accomplish certain task. The main idea behind SOAs is that "instead of building or buying monolithic software systems, in which the business logic is hard-coded, applications should be composed in a flexible way, using well-defined software services that may be distributed over the Internet" [48]. SOA enables lightweight approach to the collaboration among different organizations by exposing their internal operations as services. In the context of SOAs, service providers expose their services by using service brokers (contains directory of services), and those services can be found by service requesters. Web services represent the most-promising architecture for implementation of SOA paradigm by using the Internet as communication medium and some well-known protocols, including the Simple Object Access Protocol (SOAP) [134] for transmitting data, the Web Services Description Language (WSDL) [136] for defining services, and the Business Process Execution Language for Web Services (BPEL4WS) [49] for orchestrating services and Web Services Choreography Description Language (WS-CDL) [55] for defining services choreographies. Web services are "self-describing, open components that support rapid, low-cost composition of distributed applications" [99]. Web services can be composed in entities that support automated execution of business processes (called service compositions). A typical modeling language for representing these processes is the Business Process Modeling Notation – BPMN [88].

In this context, Model-Driven Engineering [9] [34] paradigm is of great relevance, as service compositions can be represented as software models, where such service compositions are used for realization of composite applications in service-oriented enterprise computing environments. Since a business process can be realized through a composition of services, processes of this kind are also called service compositions. However, current solutions to modeling SOA compositions have some serious drawbacks [99], such as: i) inability to abstract the business logic at the problem domain level, so that changes of the (parts of) business logic do not trigger the change of overall process composition; ii) support of the modeling of complex service compositions where one should be able to define rules of interaction between multiple business process end points in a unique way; and iii) increased flexibility and adaptivity of business processes realized as SOAs by isolating variable parts from the reusable parts of a business process and by combining the reusable parts with business rules that model the variables parts.

On the other hand, we have Business process management which "includes concepts, methods, and techniques to support the design, administration, configuration, enactment, and analysis of business processes" [146]. Thus, a business process consists of a set of activities that are performed in coordination in an organizational and technical environment. These activities jointly realize some business goals. Each business process is enacted by a single organization, but it may interact with business processes performed by other organizations [146]. Business processes are represented by business process models, where following MDE principles, models are expressed with modeling languages, which are defined by metamodels that are associated with notations of the modeling languages, often of a graphical nature. A variety of modeling languages exists for the specification of process models, and they can be classified according to their focal modeling construct, according to [151]: i) *Activity-centered;* processes as a network of tasks or activities; ii) *Process object centered*; processes as the legal sequence of state changes of the process object; and iii) *Resource centered*; process as a network of processing stations that interact with each other. Process languages appear as Graph-based languages (e.g. BPMN), Net-based languages (e.g. Petri-nets, flow nets) and Workflow Programming Languages (e.g. BPEL). So, SOAs which are usually built with services as loosely-coupled computing tasks communicating over the Internet/network can be represented with Resource-centered languages, such as BPMN [88], which represent today de-facto standard for representing business processes.

Recent research [108] has identified a lack of explicit formalism in the process modeling languages for capturing business rules. The key idea is to extract business logic contained implicitly in business process models into explicit definitions of business rules. This should enable for improving business agility, so that business processes can cope with the dynamic nature of business changes, and to accommodate dynamic logic of many different applications. This allows for the specification of business knowledge in a way that is understandable by business users, and at the same time understandable by technical users and executable by rule engines, and thus, bridging the gap between business and technology [30]. The approach which hard codes some business logic within applications cannot accommodate rapid and frequent changes of a business process without a heavy burden in terms of time and cost. Web services and business rules are complementary technologies that provide a good approach to bridge such a gap. When these two approaches are deployed in combination, applications gain strengths in ways that enhance business agility. The "loosely-coupled" approach of Web services/SOA, together with the "de-coupled" approach of business rules enables applications to better represent business logic in "explicit" format that can be more flexible and easily modified and shared across many applications.

1.1. Research goals

From this we define research problem for this thesis and that is how to enable synergy between business rules language and a business processes languages for modeling SOAs in order to achieve agile SOAs (*run-time change of a business process*). Based on the research problem we define goals of this thesis, and that is development of a methodology, language and software development environment for modeling SOAs that enable for the synergetic usage of rule and process languages. Also, this language and the software development environment, which will support the language, will be used as a research instrument of the given research problem, and in the software development environment we will evaluate our research goals. Therefore, research objectives of this integration and also this thesis are:

- Defining a methodology and a modeling language for developing rule-driven (agile) business processes and SOAs;
- Integrating of business rules and vocabularies with business process models used as designs of SOAs;
- Facilitating dynamic changes of a business process execution flow by making rules first-class citizens in business process modeling due to their declarative rule nature;
- Extracting service compositions from rule-based business process models to make those process models executable;
- Translating business rules defined by domain experts into a formal representation suitable to be used by service engineers;
- Deploying rule- and vocabulary enhanced process models onto rule and service composition engines;
- Defining conditions for interaction execution and constraints in those interactions.

In order to achieve the abovementioned goals, we defined a research methodology in this thesis, which included the following activities:

- Reviewing and analyzing the literature about the state-of-the-art in the areas of SOA, business rules, and business process modeling in order to identify research gaps and position the contribution of this work;

- Design and development of a development methodology for defining rule-based business processes and SOAs;
- Design of a modeling language and a software development environment for modeling rule-based business processes and SOAs;
- Evaluation of the modeling language with respect to its capability to model common problems in the relevant area (i.e., through workflow, message exchange, and service interaction patterns) and by using a realistic case study.

1.2. Contents per chapter

This thesis consists from seven chapters and a literature section. After the Introducion section, we given the overview of the MDE concepts and related techgooglenologies (MOF, UML, QVT, ...). In addition, we described Eclipse Modeling Framework and Graphical Modeling Framework. We also introduced business process languages, as well as the modeling and technological spaces. After that we given a description of the existing rule and policy modeling languages, their usage in Service-Oriented Architectures and modeling rules and policies, as well as for the development of Service-Oriented Architectures by using MDE principles that integrates rules and policies. We gave a review of current Service-Oriented Architectures and Web services, and analysis of integration between business processes and rules.

In the third chapter, we give a proposal for a rBPMN language concrete graphical syntax, which includes the integration of certaion BPMN elements (such as gateway) and different types of business rules (reaction, production, derivation and integrity rules). This includes a proposal for extensions of BPMN for modeling choreographies, with a reference to the common problems in choreography modeling in BPMN, as well as integration of PML policies in processes by using the appropriate metamodel. Besides the graphical synax we gave a proposal for the rBPMN language metamodel in MDE architecture. In addition, we give the integrated methodology for development of secure Service-Oriented Architectures, by using rBPMN models of business process with integration of rules (R2ML) and policies (PML) in order to support different aspects of these architectures. In addition, we gave a complete proposal for designing business processes, data and rules. We also showed a support for modeling policies in a process of development of Service-Oriented Architectures (service compositions) by using modeling and PML language. In proposed methodology we gave detailed steps that should be followed during the development of rule-based business processes.

The fourth chapter provides a detailed evaluation of rBPMN language through modeling of the four major types of service composition patterns. We gave a review of the Message Exchange patterns (MEP’s), control flow patterns, interaction patterns and patterns for the agile business processes. Through these patterns we showed expressivity of rBPMN language for modeling various parts of business processes through integration with rules. All patterns were analyzed based on the possibility of their modeling by using rBPMN language. In this chapter we gave mapping between two types of process models in rBPMN language, interaction models and interconnection models.

The fifth chapter describes the case studies for rBPMN language through several scenarios of Service-Oriented Architecture usage, i.e., service composition models. However, the possibilites of this language are not limited only to the described scenarios, but it is possible to model all of the patterns from the chapter five. Specifically, in this chapter we showed orchestration modeling on the example of the on-line product order, we also gave an example of choreography modeling in the process of the flight request and an example of modeling agile business processes in rBPMN through book buying over the Internet. This chapter also show an implementation of the application for modeling rBPMN-based processes, called rBPMN editor.

The sixth chapter gives overall description of rBPMN language evaluation based on patterns given in the chapter five. In this chapter we gave a comparative analysies of existing languages for interaction modeling patterns, control flow patterns and agility patterns, through the analysis to improve modeling of these patterns by using the rBPMN language. We also gave a review of modeling various aspects in rBPMN, constraints in modeling by using standard BPMN, as well as possibiliteis for modeing these patterns by using rules.

The last chapter gives a critical review of the results achieved during the research described in this thesis. It discusses in detail the scientific, techical and practical contributions achieved in this study. After that, we gave an analysis of possibilities of practical application of the results of this study. In the end, we gave a plan for possible future research.

2. Literature Review

This chapter surveys the state of the art in the relevant areas and introduces background knowledge that is important for understanding the concepts described in the rest of this thesis. In addition, this chapter describes business process modeling, rule languages, and the principles of model Driven Engineering.

2.1. Model Driven Engineering

Model Driven Engineering is not Model Driven Architecture [34]. MDA is an OMG standard and is a specific version of the MDE approach. Favre defines MDE as an open and integrative approach to software development which involves many technological spaces (TS) [66] in a uniform way, and MDA is only one instance of MDE implemented in a series of technologies defined by the OMG (MOF, UML, XMI).

MDA introduces a set of basic concepts, such as model, meta-model, modeling language and transformation, and recommends categorization of all models to platform-independent models (PIMs) and platform-specific models (PSMs). However, MDA is not a software development process.

A technological space is defined as a working context with a set of associated concepts, body of knowledge, tools, required skills, and possibilities [66]. It is often associated with a given user community with shared expertise, educational support, common literature and even workshop and conference meetings. Examples of technological spaces are MDA and MOF, but also Grammarware [58] and BNF, Documentware and XML, Dataware and SQL, Modelware and UML, etc.

An important aspect of MDE is that it bridges different technological spaces and integrates knowledge from different research communities. In every space, model, meta-model and transformation concepts appear at various levels of abstraction and in a way can conform to certain concepts in another technical space. For example, what is called a meta-model in Modelware, conforms to something that is called a schema in Documentware, or grammar in Grammarware, etc. In [57], MDE is defined starting from MDA by adding assignment in a process of software development and a space for model organization. Two illustrative examples of the MDE process can be found in [1] and [10].

2.1.1. Definitions of model and modeling

The origin of the word *model* can be traced to the Latin *modulus*, which means a small measure. A definition of model from [123] says that: "*a model is a representation of a concept. The representation is purposeful: the model purpose is used to abstract from the reality the irrelevant details*". Miller and Mukerji state that "*A model of a system is a description or specification of that system and its environment for some purpose. A model is often presented as a combination of drawings and text. The text may be in a modeling language or in a natural language*" [85].

Computer science uses models in several phases of software development. MDA and MDE rely on modeling and models as their basic concepts. However, there is no single definition of model that is widely accepted in all computer science. Seidewitz defines model as "*a set of statements about a system under study*" [121], and [68] defines model as an "*abstraction of (real or language-based) system allowing predictions or inferences to be made*". There are a number of other definitions, presented in [67]. This thesis uses the following definition of model: "*A model represents a part of the reality called the object system, and is expressed in a modeling language. A model provides knowledge for a certain purpose that can be interpreted in terms of the object system*" [67].

Models usually serve as specifications in traditional engineering disciplines. When software is constructed, models can be used as specifications as well. A UML model can be used for describing an existing software system (its structure and operations).

Model interpretation means mapping model elements to the elements of the object system (system under study), so that a specific value of each model expression is obtained in the object system which is under study (with a certain level of accuracy). Thus, a model interpretation gives a model a meaning associated with the object system.

Modeling languages enable to write expressions with elements in models of classes systems under study. A working software system can be based on a model that represents a certain part of reality, while the software itself can be regarded as a model.

2.1.2. Modeling principles

In the world of software engineering, modeling has a rich tradition that reaches early days of programming. More recent efforts are focused on modeling languages and tools that permit users to express the system parameters to software architects and programmers, in a way that can be uniquely mapped to a concrete programming language and then compiled for a specific operating system. UML [96] is currently the most widely accepted language for visual specification of models, which is adopted as the de facto industry standard for software modeling and standardized by the Object Management Group (OMG). UML enables development teams to describe important characteristics of systems in appropriate models. Transformations between these models are usually accomplished manually, although there are tools that can do automatic model transformation [81].

A model is used for an indirect study of reality (i.e., of an object system) [67]. Various reasons may cause this indirectness. The object system may be inaccessible, or its direct study is too expensive, or even the object system may not exist yet. In all such cases, the model plays the role of a specification of the object system. Regardless of the reasons for indirectness, the model must be a valid representation of the object system. The knowledge acquired from the model must hold for the object system. Often, this knowledge is not exact but only approximates the reality, with an acceptable degree of inaccuracy. Furthermore, the knowledge acquired from the model is initially expressed in terms of model elements. This knowledge must be interpreted and converted to knowledge in terms of the object system. The relation between a model and an object system is bi-directional and two separate relations may be considered, as Figure 1 shows. This figure is called the DDI account (DDI – Denotation, Demonstration, Interpretation), and was first introduced in [47].

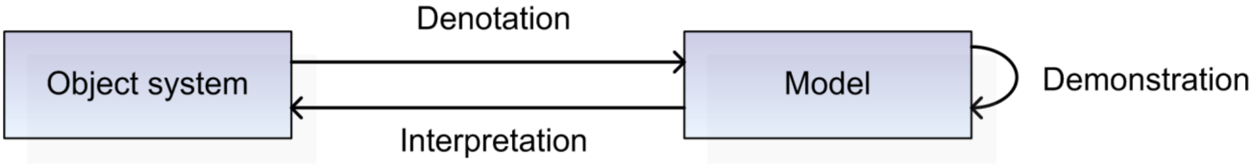


Figure 1. Relationships between an object system and its model [47]

The object system is *denoted* (represented) in a model. This denotation must preserve some characteristics of the object system to allow acquiring knowledge about it through the model. The model is used to obtain claims about the model elements. This process is known as *demonstration*. It happens only in the context of the model. Finally, the obtained results are mapped to the object system. This mapping is called *interpretation*. The knowledge obtained from the model must be verifiable against the object system. If the results obtained from the model do not meet the empirical evidence obtained from the reality, then the model is invalid with respect to the object system.

Literature usually depicts only one relation between a model and its object system. Various names for the relation are used: *ModelOf*, *RepresentationOf*, *RepresentedIn*, *ModeledBy*, etc. *ModelOf* relation will be used in the remaining part of the thesis, because it accumulates two other relations: *Denotation* and *Interpretation.*

2.1.3. Meta-models and meta-modeling

As the name suggests, meta-modeling is a modeling activity. Similarly, the product of meta-modeling, called a *meta-model*, is a model. If an entity is a model, we have to be able to clearly identify its object system. A meta-model is a model of the conceptual foundation of a language, consisting of a set of basic concepts, and a set of rules determining the set of possible models denotable in that language [33]. Therefore, a meta-model describes what models in that language can express. Based on this, we can conclude that a meta-model is a model of models expressed in a given modeling language [121]. Since a meta-model itself is a model, it is also represented in some modeling language. One modeling language can have more than one meta-model, each one represented in a different modeling language. Of special interest is the case when the meta-model of a modeling language uses the same modeling language. In that case, expressions in the meta-model are represented in the same language that describes the meta-model. This meta-model is called *reflexive meta-model*. *Minimally reflexive meta-model* uses a minimum number of modeling language elements (for that meta-model purpose). Since this meta-model is defined as reflexive, there is no need for upper levels, because it defines itself with its own concepts.

Generally, there is a *modelOf* relation between a meta-model and its object system; it is a modeling language. An *instanceOf* relation between a meta-model and a model often replaces it. Indeed, they coincide between the same entities but are different in nature. The grammar of some programming language possesses characteristics of all words (and sentences) which that language can contain. So, we can take a language grammar as a model of that language (an example of such a grammar is the Extended Backus-Naur Form, EBNF). In the case of a modeling language, the model of this language is its meta-model. The relation between a model written in some language and its meta-model is called *conformantTo* [35]. This relation is defined as a composition of two relations: *elementOf*, denoting the membership of a model to a language, and *representationOf*, denoting the relation between an object system and its model. An example of a meta-model, a model, and an *instanceOf* relation is shown in Figure 2.

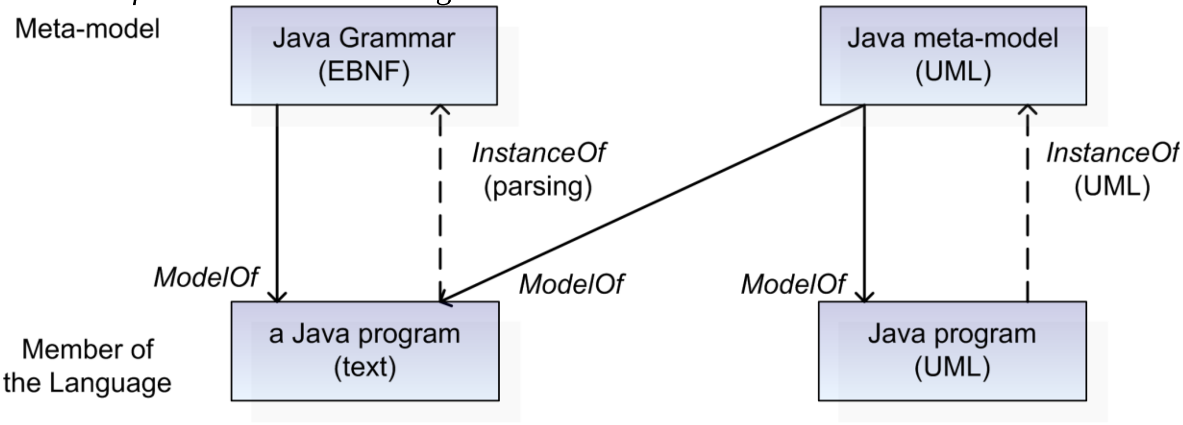


Figure 2. Example of meta-models, models and *instanceOf* relations [67]

An important difference between the two relations is observed when the language-dependent nature of *instanceOf* is considered. Let us assume that we define another meta-model of the Java language expressed in UML (see Figure 2). The UML meta-model may contain a class called Method. The knowledge we obtain is that there is a set of methods in every Java program that has a certain structure. We must be able to identify methods in the source program and to recognize their structure according to the definition of the Method class. It is the consequence of the *ModelOf* relation that exists between the Java meta-model and a Java program. However, we cannot consider the Java program as an instance of the UML model in the same way as we did it for the Java grammar. An instance of the UML model is defined according to the semantics of UML, and is a set of objects. This instance is a representation of the Java program and is a different entity. The UML model of Java is also a model of the Java program represented in UML. In addition, there is an *instanceOf* relation between these entities governed by the UML semantics. Much like the relation between a source program and its grammar, this *instanceOf* relation helps us interpret the knowledge from the UML model in terms of the Java program represented in UML. These two *instanceOf* relations are different. The first one is defined for the parsing process. The second one relies on the UML semantics. There is no direct language-specific *instanceOf* relation between a source program in Java and its UML model. However, the latter is a model of the former, although we cannot trace the knowledge from the model to the object system via an *instanceOf* relation.

In summary, we can say that *instanceOf* relation exists between a class and its members and supports the interpretation of the knowledge obtained from the class definition in terms of class members. In that case, we also have a *ModelOf* relation between the class definition and class members.

2.1.4. Meta-modeling architecture

A meta-modeling activity can be applied to specify a modeling hierarchy that assumes a multi-level organization, called *meta-modeling architecture*. Figure 3 shows an example of this architecture.

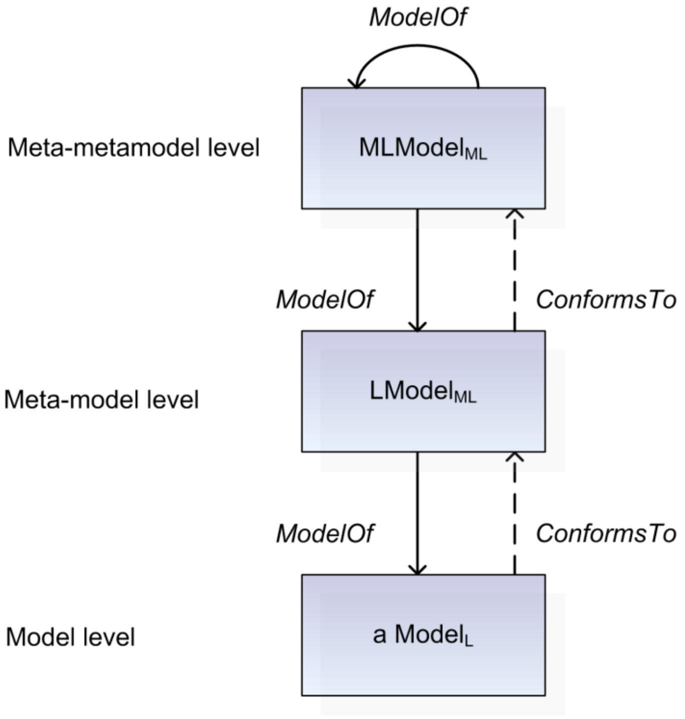


Figure 3. Meta-modeling architecture [67]

The *ConformsTo* relation means that a model is constrained by the rules defined in its meta-model. At the bottom level of this architecture, we have models expressed in various modeling languages. This level is called the *model level*. An example model in this level is $Model_L$ written in a modeling language *L*. We can build a model of *L* (that is, a meta-model) $LModel_{ML}$ expressed in another language, called *Meta-language* (ML). Models of the languages used in the model level form the second level in the stack. It is called the *meta-model level*. There is a *ModelOf* relation between the meta-model of a language and models expressed in that language. We can apply the same approach to the models at the meta-model level. The models of the languages that express meta-models form the third level, called the *meta-metamodel level*. At the third level of the meta-modeling architecture shown in Figure 3, the model *MLModel* is expressed in the *ML* language itself. In this way, the top level contains a self-reflective model. It is expressed in the language that is modeled by that model. The intuition behind this is the following. At the meta-model level, we have models of modeling languages expressed in *ML*. However, *ML* is a modeling language itself, and therefore it should be possible to apply *ML* itself to express its model.

Examples of technologies that rely on meta-modeling architecture are Meta Object Facility (MOF), section 2.1.5.1, and Eclipse Modeling Framework (EMF), section 2.1.5.5.

An example of the relation between a model and its meta-model in Figure 4 that represents the *meta* relations between a Petri Net model and a simplified Petri Net meta-model, represented in UML. *Meta* relation, associates each element of a model with the meta-model element it instantiates.

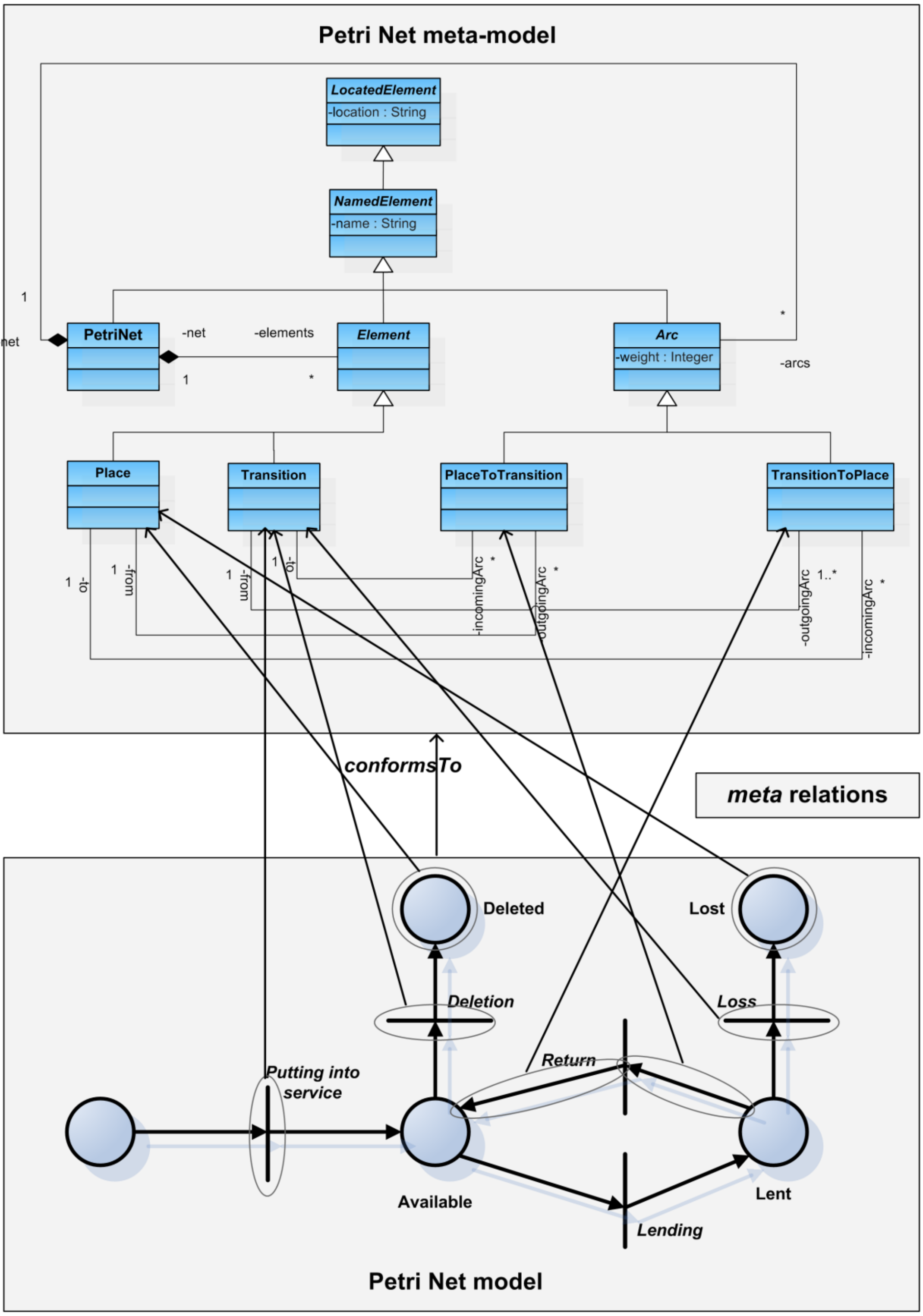


Figure 4. Meta relations between Petri Net model and meta-model [4] [36]

As any other model, a Petri Net model network is composed of a certain number of different elements. In the context of Petri nets, these elements conform to *places*, *transitions* and *arcs*, and they constitute a model. These different elements, together with the way they are connected, conform to the Petri Net meta-model. In the same way, each model conforms to its meta-model. This relation associates each model element with a meta-model element that it instantinates. In addition, the meta-model itself can conform to some meta-metamodel (as it is shown in Figure 3, *MLModel*$_{ML}$).

2.1.5. Model Driven Architecture

The Model Driven Architecture (MDA) defines an approach to specifying Information Technology (IT) systems and that separates the specification of functionality from the specification of the implementation of that functionality on a specific technology platform [85]. The MDA approach and the standards that it supports enable for a model that determines some system functionality to be realized on multiple platforms through additional standards for mapping. The MDA is specified by the OMG consortium[1] in a series of standards: Unified Modeling Language (UML), Meta-Object Facility (MOF), Common Warehouse Metamodel (CWM), etc. An illustration of the MDA idea is shown in Figure 5.

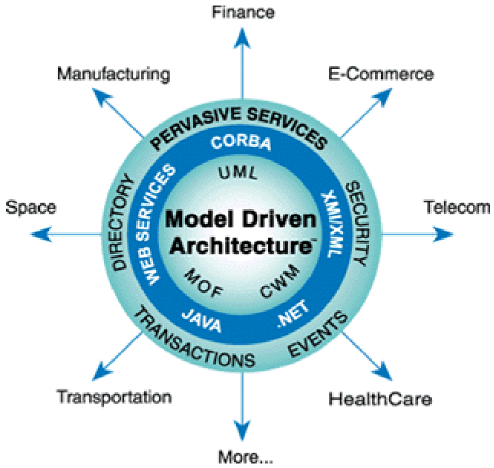


Figure 5. Model Driven Architecture (OMG)

Model is the most basic element of the MDA. There are several definitions of the term "model" (see section 2.1.1), and the most general one is that a model is a simplified view of reality [122]. Each model itself is defined for some domain, and then it is transformed to models that can be executed on a specific platform. A basic assumption of MDA is that a unique model underlies each information system. Such a model does not depend on a potential implementation platform, on which the corresponding application can be run. In other words, the system requirements can be specified as a *Computation Independent Model* (CIM) [85]. The model defined at this level is sometimes also called the *domain model* or the *business model*. It does not depend on how the system is implemented. In software engineering, a domain model is specified by the domain experts. *Platform Independent Model* (PIM) can be also used to describe a system. It is lower-level and more specific than CIM in terms of being a computation-related model, but it does not include characteristics of specific computer platforms. To get a model that takes into account some target platform specifics, i.e., a *Platform Specific Model* (PSM), we need to define certain transformations that transform the corresponding PIM to the desired PSM. Each PSM includes information about some software implementation details (such as the programming language and operating system) and the hardware platform. Code generation is done by additional translation from the PSM into a certain programming language.

The MDA is based on four-layer metamodeling architecture shown in. The standards supporting the four-layer MDA architecture are:

- Meta-Object Facility (MOF);
- Unified Modeling Language (UML);

[1] The Object Management Group, http://www.omg.org/.

- XML Metadata Interchange (XMI).

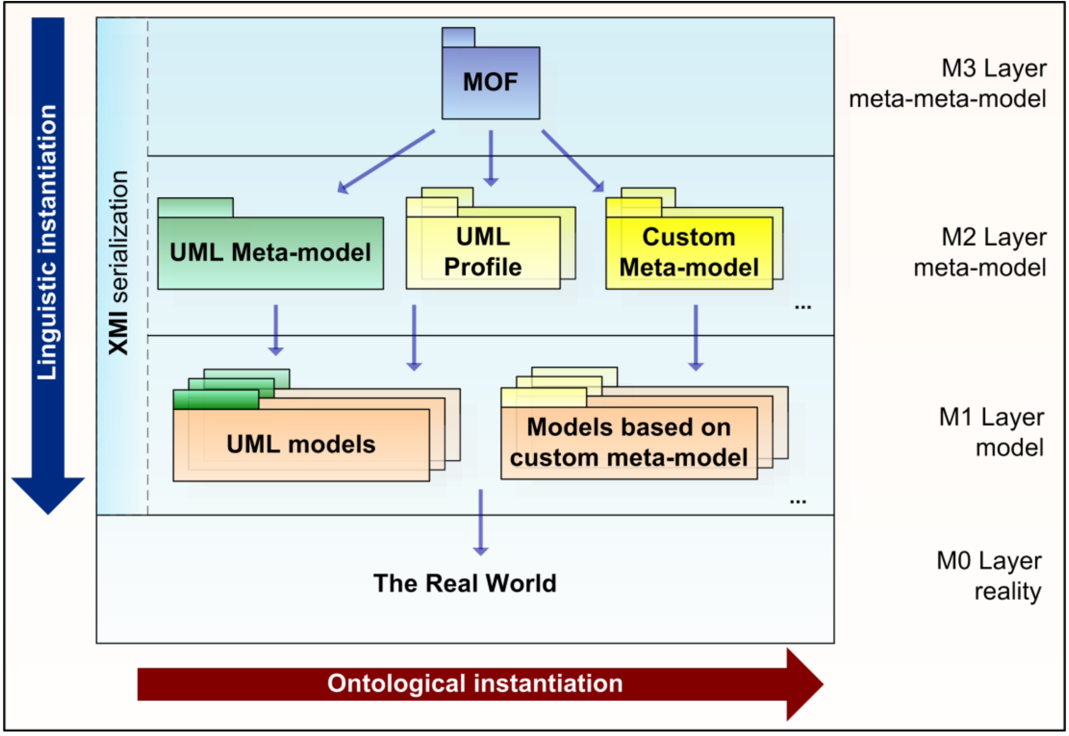


Figure 6. The four-layer Model Driven Architecture and its orthogonal *instanceOf* relations: linguistic and ontological [37]

On top of this architecture, at the M3 level, is a reflexive meta-metamodel, which is called MOF. It is an abstract self-defined language and a framework for specifying, constructing, and managing technologically independent meta-models. It is a basis for defining any modeling language, such as UML or MOF itself. MOF also defines a backbone for the implementation of a metadata (i.e., model) repository described by meta-models. The rationale for having these four levels with one common meta-metamodel is to enable both the use and generic managing of many models and meta-models, and to support their extensibility and integration.

All meta-models, standard and custom (user-defined), that are defined in MOF are placed at the M2 level. One of these meta-models is UML, which is a language for specifying, visualizing, and documenting software systems. The basic UML concepts (e.g. Class, Association, etc) can be extended in UML profiles in order to adapt UML for specific needs. Models of the real world, which are represented by concepts of a meta-model from the M2 level, are at the M1 level of the MDA four-level architecture. The bottom layer is the instance layer (M0). At the M0 level are things from the real world that are modeled at the M1 level. For example, the MOF Class concept (from the M3 level) can be used for defining the UML Class concept (M2), which further defines the Student concept (M1). The Student concept is an abstraction of a real thing *student*.

One can ask the question: what layer contains abstractions of a certain model? If we consider classes, their instances in UML are objects. However, objects are defined at the M2 level in the UML meta-model, which means that their instances are located in the M1 layer. Since even objects themselves model concrete (singular) real-world things, this explanation can be considered true. In [5] it is said that there are two types of instantiation in meta-modeling: *linguistic* and *ontological*. Linguistic instantiation is interpreted in the MDA in an ordinary way - it means that a UML class is an instance of the meta-class from the UML meta-model. However, one class in some domain has instances that are objects. The relation between objects and class is an ontological instantiation relation. This kind of instantiation connects abstractions located at the same linguistic layer. According to this interpretation, at the M0 layer are things from real world (instances) and abstract concepts about thing groups (classes). UML 2.0 and MOF 2.0 emphasize the linguistic dimension. Ontological levels exist at the M1 level, but the meta-model border does not explicitly separate them. This is based on an altered perception of the MDA four-layer architecture, as originally class instances have been located in the M0 layer.

XML Metadata Interchange (XMI) is the standard that defines mappings of MDA-based meta-metamodels, meta-models, and models onto XML documents and XML Schemas [93]. Since XML is widely supported by many software tools, it empowers XMI to enable better exchange of meta-metamodels, models, and models (see section 2.1.5.3).

2.1.5.1. Meta-Object Facility (MOF)

Meta-Object Facility (MOF) [92] in its current version (2.0) represents an adaptation of the UML core. MOF is a minimal set of concepts that can be used to define other modeling languages. It is similar (but not identical) to the part of UML used in structural modeling. In the latest version of MOF (2.0), concepts, as well as UML Superstructure concepts [96], are derived from the concepts defined in the UML Infrastructure standard [96].

Figure 7 shows meta-models that depend on the UML core package. UML Core package defines the basic concepts that are used in modeling (e.g. Elements, Relationships, and Classifiers). In MOF 2.0, there are two meta-metamodels:

- *Essential MOF* (EMOF) - represents a basic package that has a minimal number of elements for modeling (e.g., `Class`, `Property`, and `Operation`).
- *Complete MOF* (CMOF) - more complex, includes EMOF, but also enables a higher expressivity, with concepts such as `Link`, `Argument`, `Extent`, and `Factory`.

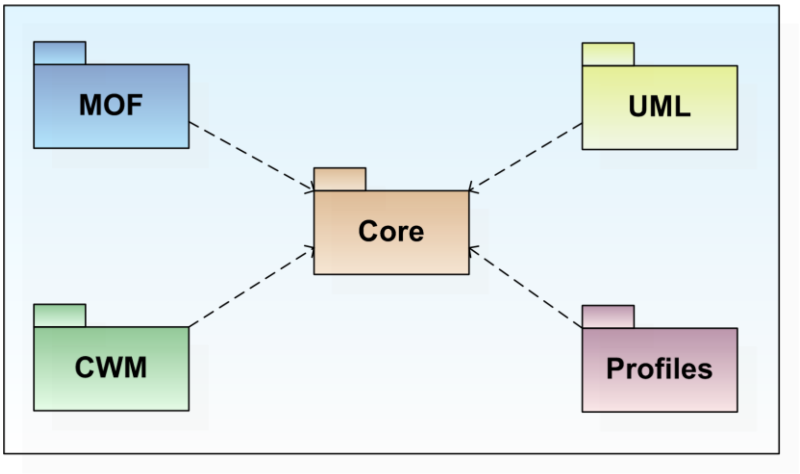


Figure 7. Core package as the common kernel [37]

The main four modeling concepts in MOF are:

- `Class` - models MOF meta-objects, concepts which are entities in meta-models (e.g., UML `Class`, `Attribute` and `Association`);
- `Association` - models binary relationships (e.g., UML and MOF superclass);
- `Package` - modularizes other concepts, i.e. groups similar concepts;
- `DataType` - models primitive types (e.g., `String` and `Integer`).

In the root of the MOF hierarchy is the `Element` concept. It classifies elementary, atomic model elements. All other concepts in MOF inherit from this concept.

2.1.5.2. Unified Modeling Language (UML)

Unified Modeling Language (UML) is a language for specifying, visualizing, and documenting software systems, as well as for modeling business and other non-software systems [96]. UML enables diagram construction, which models a system by describing conceptual things (e.g., a business process) and concrete things (e.g., software components). UML is not limited only to software engineering domain; it can be used in other areas, such as: banking, health care, defense, etc. UML is often identified as a graphical notation, which was true for its initial versions. Recently, UML is recognized more like a language independent from a graphical notation rather than a graphical notation itself.

The basic building block of UML is a diagram. There are several types of diagrams for specific purposes (e.g., time diagrams) and a few for generic use (e.g., class diagrams). UML version 2.0 defines the following types of diagrams:

- use case diagram;
- class diagram;
- behavior diagrams:
  - activity diagram;
  - statechart diagram;
- interaction diagrams:
  - sequence diagram;
  - collaboration diagram;
- implementation diagram;
  - component diagram;
  - deployment diagram.

When UML is applied to software, it represents a bridge between the original idea for some software and its implementation [104]. UML also provides a possibility for collecting specific requirements for some specific system.

UML as a graphical notation is not a software process; it is designed for use in a process of software development and it possesses all characteristics that enable it to be a part of a software development process. Since main UML diagram concepts are defined in the *Superstructure* package of the UML specification that includes basic concepts of the UML core [96], it can be said that MOF and UML are very similar.

2.1.5.2.1. UML Profiles

UML Profiles combine concept *stereotypes*, *tagged values*, and *constraints* in order to define a precise UML dialect for a specific purpose. This means that it is possible to create new types of elements for modeling by extending existing elements. When new elements are created, it is possible to add them to existing UML tools. With profiles, classes can be extended with stereotypes that represent predefined classes with certain methods and attributes. For example, Figure 8 shows one such a stereotype - EJBEntityBean.

A UML Profile definition in the context of the MDA four-layer meta-modeling architecture means extending UML at the meta-model layer (M2). Tagged values are defined as stereotype attributes (in Figure 8 tagged values of `EJBEntityBean` are *IsReadOnly*, *DataSource*, etc.). It is possible to define constraints that additionally refine the semantics of the modeling element they are attached to. They can be attached to each stereotype using OCL (Object Constraint Language) or the English language (i.e. natural language) comments, in order to precisely define the stereotype's semantics.

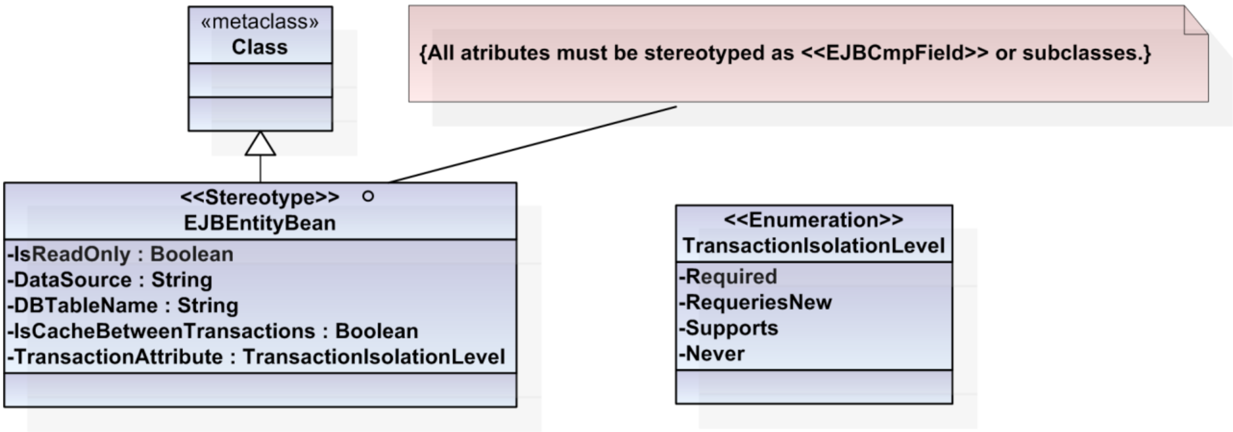

«metaclass»
Class
{All atributes must be stereotyped as <<EJBCmpField>> or subclasses.}
<<Stereotype>>
EJBEntityBean
-IsReadOnly : Boolean
-DataSource : String
-DBTableName : String
-IsCacheBetweenTransactions : Boolean
-TransactionAttribute : TransactionIsolationLevel
<<Enumeration>>
TransactionIsolationLevel
-Required
-RequeriesNew
-Supports
-Never

Figure 8. An example UML Profile for Enterprise applications in Java

So far, many important UML Profiles have been developed. Some UML Profiles are adopted by OMG, such as Enterprise Application Integration [132] and UML Profile for MOF [133]. In addition to these formal specifications, there are several well-known UML Profiles widely accepted by software engineers, such as UML Profile for building Web application developed by Jim Conallen [19].

2.1.5.3. XML Metadata Interchange (XMI)

XML Metadata Interchange (XMI) is an XML-based standard for sharing meta-data in the MDA [93]. XMI is defined by XML, using two XML Schemas:

- XML Schema for MOF meta-models;
- XML Schema for UML models.

The first one defines the syntax for sharing both MOF-based meta-models and the MOF definition itself. Since UML is a modeling language that developers use for describing various models, it is obvious that there is a need for an XML Schema for exchanging UML models. In fact, there is a standardized one called the UML XMI Schema. The UML tools such as IBM/Rational Rose, Poseidon for UML, Together, etc. support it, but some researchers report that we always loose some information when sharing UML models between two UML tools [126]. OMG has released several versions of the XMI standard: 1.0, 1.1, 1.2 and 2.0, and the latest version is 2.1.

Figure 9 shows the relationship between UML models and XMI files.

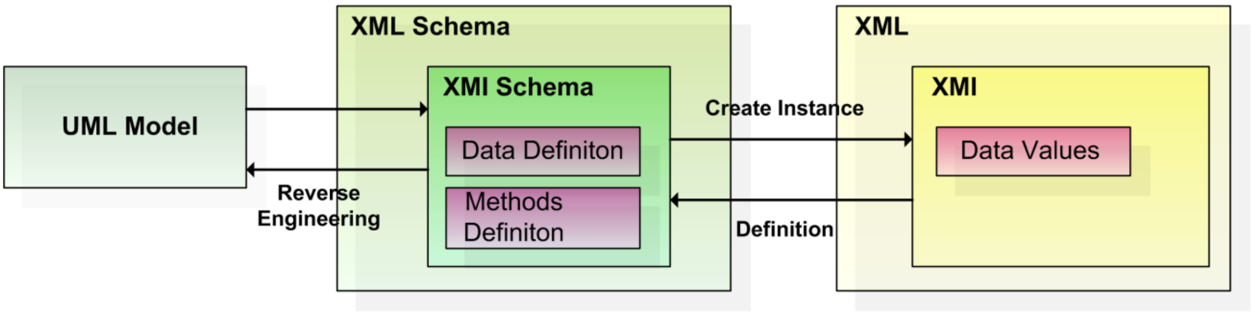
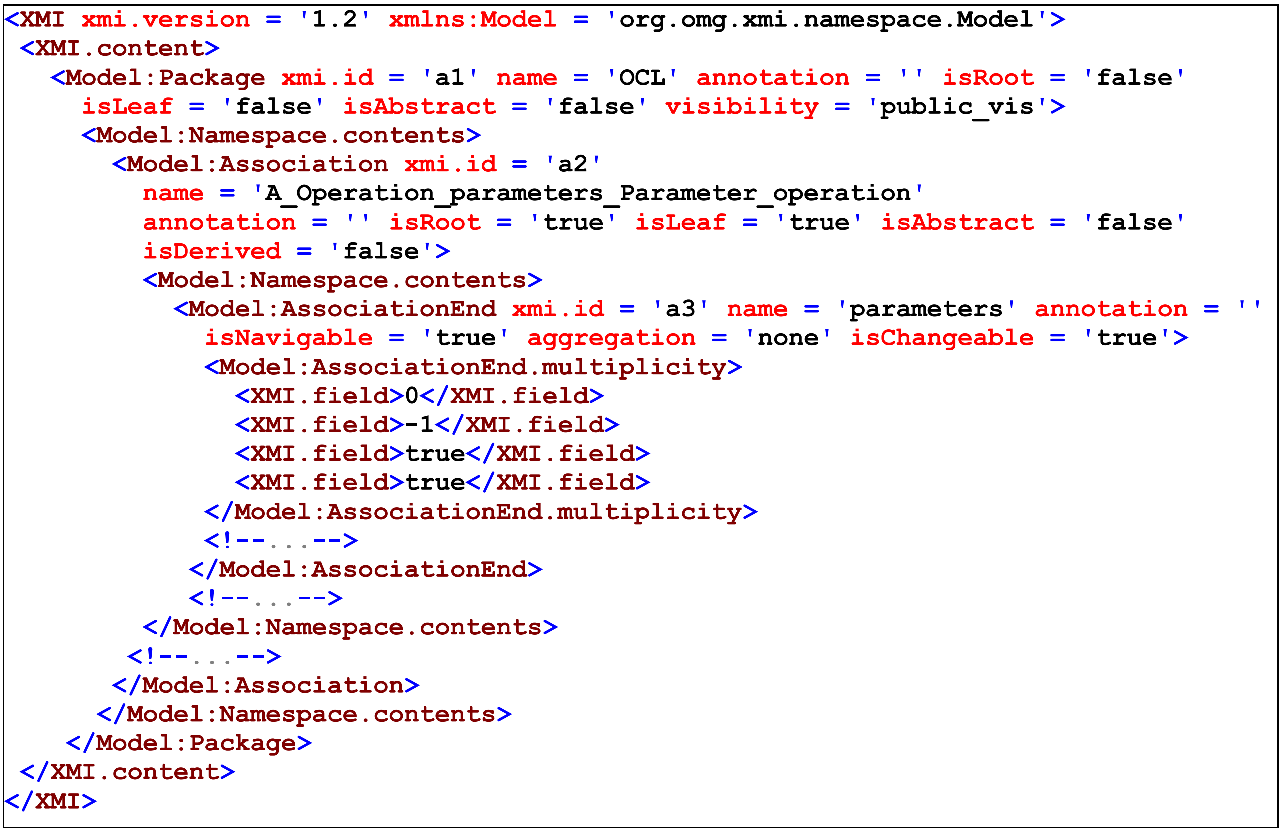


Figure 9. Relationship between UML, XML Schema and XMI

Since there is a set of rules for mapping UML and MOF models to XML Schema, it is possible to create XML Schema for every UML model. Objects as instances of such a model can be interchanged conforming to these schemas. An XML Schema can be created for any MOF-based meta-model.

An example of an XMI file (in version 1.2) is shown in Figure 10.



```
<XMI xmi.version = '1.2' xmlns:Model = 'org.omg.xmi.namespace.Model'>
 <XMI.content>
   <Model:Package xmi.id = 'a1' name = 'OCL' annotation = '' isRoot = 'false'
     isLeaf = 'false' isAbstract = 'false' visibility = 'public_vis'>
     <Model:Namespace.contents>
       <Model:Association xmi.id = 'a2'
         name = 'A_Operation_parameters_Parameter_operation'
         annotation = '' isRoot = 'true' isLeaf = 'true' isAbstract = 'false'
         isDerived = 'false'>
         <Model:Namespace.contents>
           <Model:AssociationEnd xmi.id = 'a3' name = 'parameters' annotation = ''
             isNavigable = 'true' aggregation = 'none' isChangeable = 'true'>
             <Model:AssociationEnd.multiplicity>
               <XMI.field>0</XMI.field>
               <XMI.field>-1</XMI.field>
               <XMI.field>true</XMI.field>
               <XMI.field>true</XMI.field>
             </Model:AssociationEnd.multiplicity>
             <!--...-->
           </Model:AssociationEnd>
           <!--...-->
         </Model:Namespace.contents>
        <!--...-->
       </Model:Association>
      </Model:Namespace.contents>
    </Model:Package>
 </XMI.content>
</XMI>
```

Figure 10. An excerpt from the MOF XMI document representing the OCL meta-model

2.1.5.4. Object Constraint Language (OCL)

Object Constraint Language 2.0 (OCL) as an addition to the UML 2.0 specification. It provides a way for expressing constraints and logic in models. OCL represents a language for defining integrity rules. It is not new in UML 2.0; OCL was first introduced in UML 1.4. However, from UML version 2.0 it is formalized by using MOF 2.0 and UML 2.0, which is defined in the UML OCL2 specification [94]. OCL is just what its name says: a language. It has its syntax and semantics defined by the UML language, and it has keywords. By its design, OCL represents just a query language, and it cannot change a model in any way [104].

OCL can be used for expressing: different pre- and post-conditions, invariants (constraints that always must be true), constraint conditions, and results of model executing. It can be used anywhere in UML, and it is usually associated to a class by using a comment (annotation). When an OCL expression is evaluated, the result is temporary. This means that the associated class, i.e., its concrete instances (objects), cannot change its condition during the expression evaluation.

OCL has four basic data types: `Boolean`, `Integer`, `Real` and `String`. Each OCL expression must have a context. The context can often be identified by where the expression is written. For example, a constraint can be attached to an element by using a comment. The context of a class instance can be referred to by using the keyword `self`. For example, if we have a constraint on the class `Student` that says: "a student's average grade (attribute *average* of type `Real`), must always be greater than 5.0", an OCL expression can be attached to the class `Student` by using a comment and by referring to the average in this way: `self.average > 5.0`.

OCL also includes constraints on methods and attributes, as well as different types of conditions, and possesses a possibility (methods) for manipulating data collections.

2.1.5.5. Eclipse Modeling Framework (EMF)

Eclipse Modeling Framework (EMF) is a conceptual modeling framework for Eclipse [124]. Eclipse is an open-source project lead by a consortium of companies, IBM being among them, with the goal to provide a highly integrative tool platform. Its current version is 2.6.1 (September 2010). Eclipse includes a core and generic environment for tool integration and a Java environment for development that is built by using that core. Other projects use the basic core to support different types of tools and development environments. The projects in Eclipse are implemented in Java and can be run on most operating systems.

The Central part of the EMF-based modeling is a model, which includes a set of elements defined by UML and its standard notation. It is a UML class diagram in the first place. In the EMF, a model is not that general and high-level as it is usually assumed.

The EMF does not require a complete, distinct methodology or some sophisticated tools for modeling. Eclipse Java Development tools are the only tools that are really needed. EMF connects modeling concepts directly with their implementations, thus bringing Eclipse and Java programmers closer, which results in modeling possibilities that are easy to learn.

2.1.5.5.1. Basic concepts of the Eclipse Modeling Framework

EMF is a Java-based environment for development of tools and other applications based on a structured model. It enables for developing a complete model for an application by using UML diagrams. This model can be used only for documentation, or it can be used as input for generating a part of an application or the complete application. This class of modeling usually requires expensive tools for object-oriented analysis and design. EMF is often used as a model handler, by model transformation tools. An important characteristic of the EMF is that it offers a "low entry price" because it requires only a small portion of UML modeling (classes and their attributes and relations), i.e. only a graphical modeling tool. EMF uses XMI for storing model definitions. To create such a document, there are four options:

1. creation of an XMI document, directly, by using an XML or text editor;
2. export of an XMI document from modeling tools (such as IBM Rational Rose);
3. annotation of Java interfaces with model attributes;
4. use of XML Schema to describe the form of model serialization.

The first and third approaches require knowledge of XML and Java, respectively, which is good if the developer is familiar with these technologies. The second approach is preferred if we use a modeling tool. The last approach is suitable for creating applications that must read or write some XML content to a file.

EMF consists of three fundamental pieces: Core, EMF.Edit and EMF.Codegen. Core provides a basic support for generating and executing classes implemented in Java for a model. It includes a meta model (ECore) for describing models and runtime support for the models including change notification, persistence support with default XMI serialization, and an efficient reflective API for manipulating EMF objects generically. EMF.Edit includes generic reusable classes for building editors for EMF models and extends the Core by adding support for generating adapter classes that enable preview and work with the model, as well as a basic (visual) editor for the model. It also has a command framework, including a set of generic command implementation classes for building editors that support fully automatic undo and redo. The EMF code generation facility (EMF.Codegen) is capable of generating everything needed to build a complete editor for an EMF model. It includes a GUI from which generation options can be specified, and generators can be invoked. The generation facility leverages the JDT (Java Development Tooling) component of Eclipse.

An overview of possibilities and the process of creating an ECore model are shown in Figure 11.

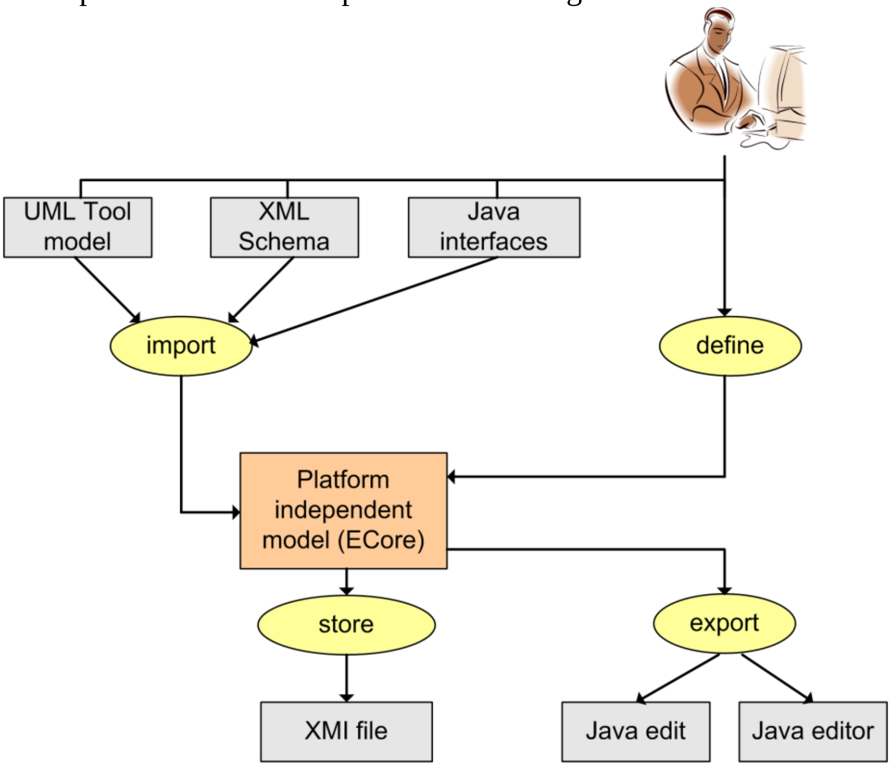


Figure 11. Creating a platform independent ECore model

EMF also supports three levels of code generation (from the model). It can generate a model, which is Java interfaces and implementation classes of all of the model classes, adapter, which adapts the model classes for editing and display (called ItemProviders), and editor, which is actually a structured editor for an EMF model.

2.1.5.5.2. ECore modeling concepts

The model used to represent EMF models is called ECore. ECore is itself an EMF model, so we can say that it is the meta-model to itself and is usually used to specify platform independent models. It is actually also a meta-metamodel. There is often a misunderstanding about meta-metamodels, but this concept is actually very simple. A meta-metamodel is just a model of another model, and if that other model is a meta-model to itself, then meta-model is actually meta-metamodel (this concept can recursively go to meta-meta-metamodels, but ECore puts a limit here, because it is described by itself). Figure 2.34 shows the ECore model with its core elements (attributes, relations and operations).

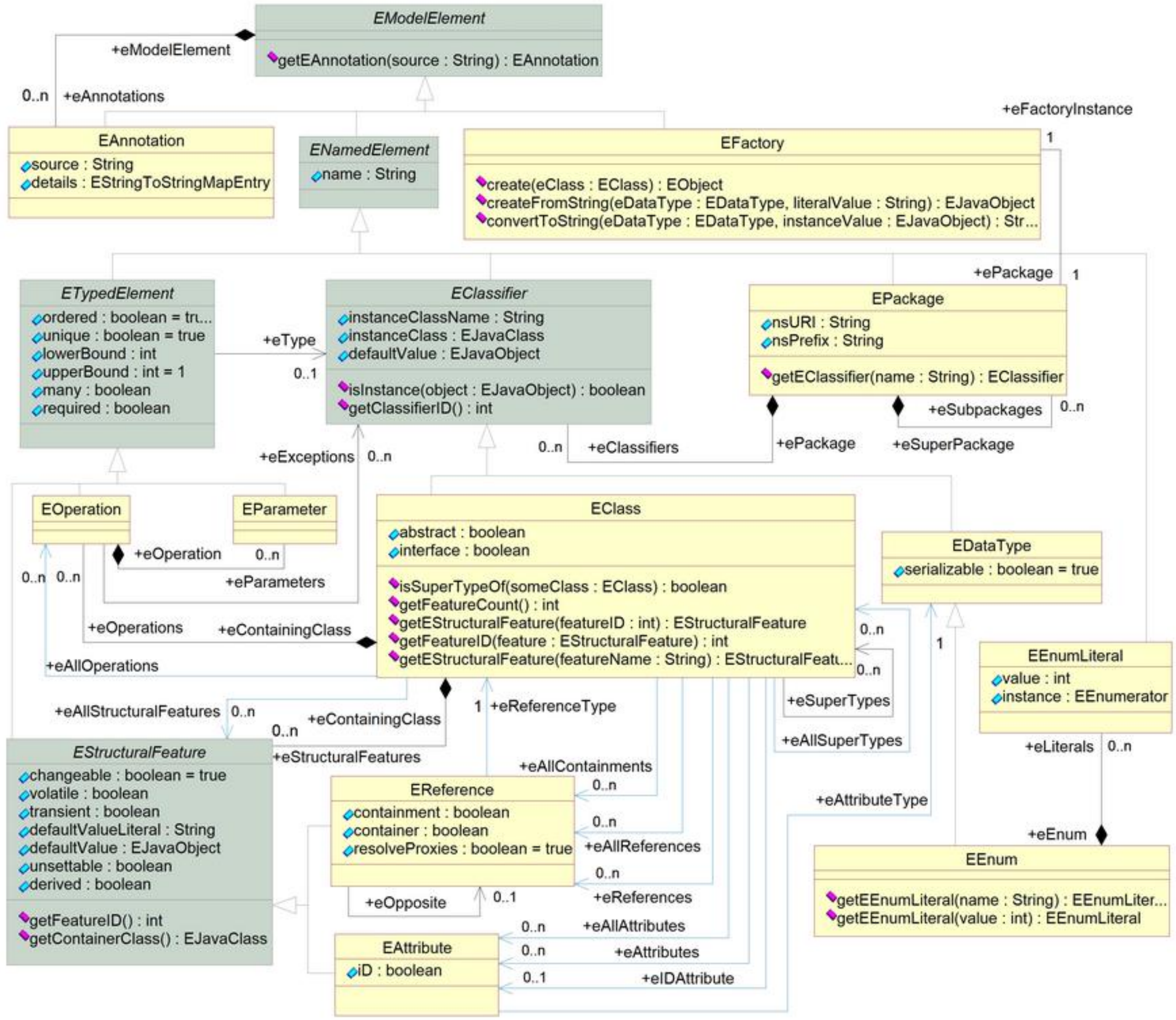


Figure 12. ECore model core elements[2]

From the above diagram, we can see that there are four basic ECore classes that are needed to represent a model:

1. `EClass` - used for representing a modeled class. It has a name, zero or more attributes, and zero or more references.
2. `EAttribute` - used for representing a modeled attribute. Attributes have a name and a type.
3. `EReference` - used for representing an association end between classes. It has a name, a boolean attribute that indicates if it implies containment, or a destination reference type, which is another class.
4. `EDataType` - used for representing attribute types. This type can be primitive, such as `int` or `float`, or object type, such as `java.util.Date`.

[2] http://dev.eclipse.org/viewcvs/index.cgi/org.eclipse.emf/org.eclipse.emf/plugins/org.eclipse.emf.ecore/model/Ecore.mdl?root=Modeling_Project&view=log

Here we can see that ECore is a small and simple subset of the UML. The standard UML supports a more sophisticated modeling than it is the case with EMF core (e.g., UML supports much more complex specification of behavior).

The class instances defined in ECore are used for describing the application model class structure. When the classes defined in ECore are extended (inherited) for defining a specific application model, it is called the basic (or core) model. Figure 13 shows an example UML class named `Student`, with two `String` attributes (name and surname). EMF generates a corresponding ECore class (such as `EClass`) and represents it with a Java interface and an appropriate implementation class. The `EClass` for the `Student` class is mapped to a Java interface:

```
public interface Student ...
```

and to an appropriate implementation class:

```
public class StudentImpl extends ... implements Student {…}
```

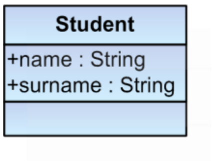


Figure 13. UML class – Student

This separation of interfaces and implementation is a choice that is enforced in the EMF design. The reason for this separation is to stay in line with good programming practices. It is important to notice that the generated interface directly inherits the `EObject` interface:

```
public interface Student extends EObject { ... }
```

`EObject` is an equivalent to the `java.lang.Object` class, which represents the root class for all modeled objects. By inheriting from `EObject`, the following behaviors are inherited:

- *eClass()* - returns the object's meta-object (`EClass`);
- *eContainer()* and *eResource()* return the object's container and resource;
- *eGet()*, *eSet()* and *eUnset()* provide an API for reflexive access to objects.

For each attribute in an interface, two appropriate set/get method signatures are created:

```
String getName();
void setName(String value);
```

Using its notification system (`EObject` interface inherits `Notification` interface), EMF enables for a simple implementation of relations (references) between classes, i.e., objects. In addition, ECore has the possibility to work with methods through its behavior attributes (`EOperation` and `EParameter` classes, which represent methods and parameters, respectively), as well as with packages (and element factories), data types and enumeration types.

The ECore class hierarchy is shown in Figure 14.

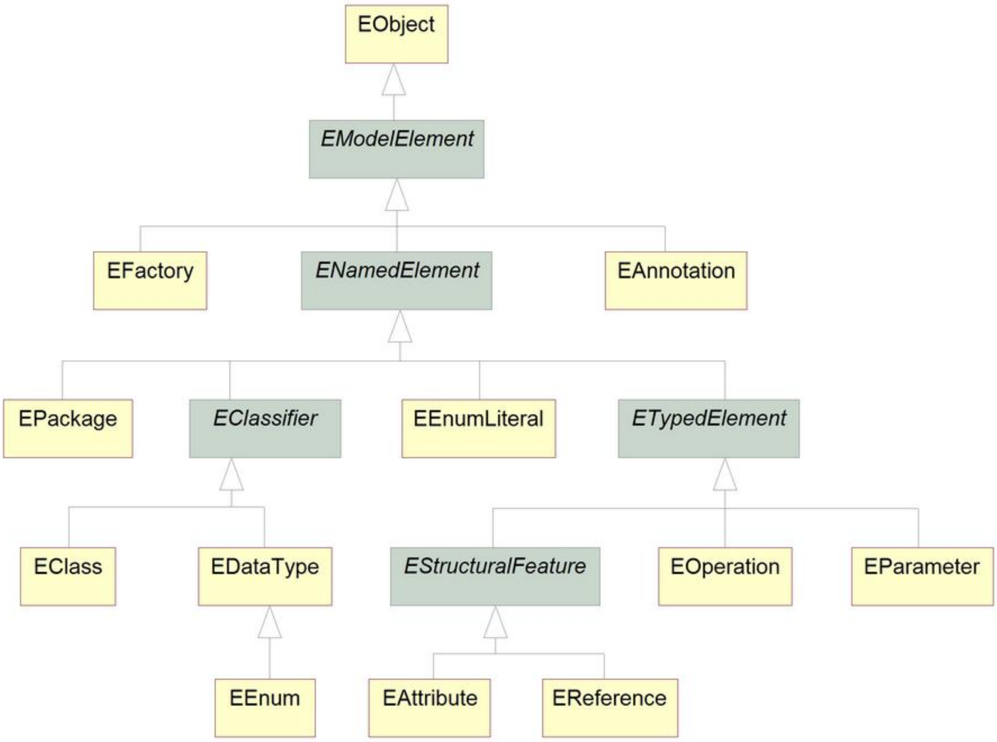


Figure 14. ECore class hierarchy

Each ECore model is stored in an XMI file and starts with the definition of a package that contains all other elements (classes, attributes, etc.). An example of an EMF XMI file is shown in Figure 15.

```
<ecore:EPackage
    xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
    name="package">
</ecore:EPackage>
```

Figure 15. EMF XMI file that contains one (*ECore*) package

`EPackage` represents a package definition, where *xmi:version* XMI version (OMG), *xmlns:xmi* and *xmlns:ecore* define namespaces for two XML Schemas that are used, and *name* is the package name.

A class (for example `Student`) is defined by the `eClassifiers` tag and the meta-reference *xsi:type="ecore:EClass"*. Attributes are represented as `eStructuralFeatures` with `EAttribute`. An example of a class defined with two attributes in an XMI file is shown in Figure 16

```
<eClassifiers xsi:type="ecore:EClass" name="Student">
      <eStructuralFeatures xsi:type="ecore:EAttribute"
         name="name" lowerBound="1"
         eType="ecore:EDataType http://www.eclipse.org/emf/2002/#//EString"/>
      <eStructuralFeatures xsi:type="ecore:EAttribute" name="surname"
         eType="ecore:EDataType http://www.eclipse.org/emf/2002/#//EString"/>
</eClassifiers>
```

Figure 16. XMI file with the ECore class serialized from the UML class Student (see Figure 13)

2.1.5.6. Graphical Modeling Framework (GMF)

While EMF enables generation of code from models, the Graphical Editor Framework (GEF) [28] allows developers to create graphical editors for models in a specific domain. EMF does not support generation of graphical editors, so GEF must be used to create basic and advanced editor functionalities by hand-coding all elements of such editors, such as figures and editor commands. Every diagram is supported by model data, which also includes layout information (e.g., figure positions). GEF offer viewers that can be used in Eclipse to display and edit models graphically. It is based on a Model-View-Controller (MVC) architecture, where the controller is located between the model and the view. It is used to observe the model and update the view to show changes made in a model. GEF have two types of viewers, graphical and tree-based.

The Eclipse Graphical Modeling Framework (GMF) [29] is made to bridge a gap between GEF and EMF. It is a framework for building graphical editors: such are UML or business process modeling editors. GMF has two main components, tooling and runtime. Tooling is used to create or edit models by describing notational and tooling aspects of a graphical editor. GMF framework also has a generator that can create a complete graphical editor implementation. Generated plug-in depends on a GMF runtime to produce a graphical editor. Creation of a GMF-based graphical editor is shown in Figure 17.

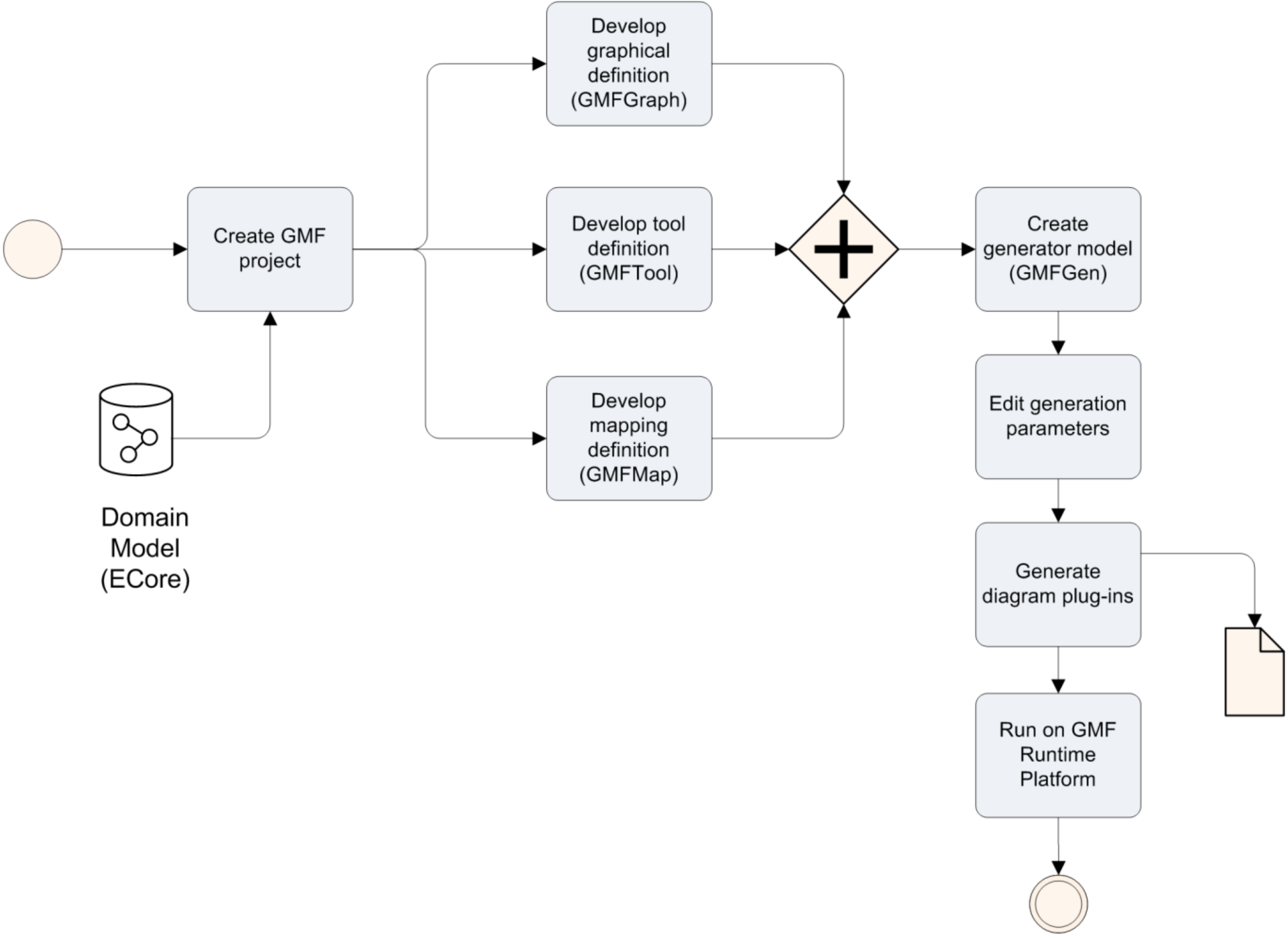

Develop graphical definition (GMFGraph)
Create GMF project
Develop tool definition (GMFTool)
Create generator model (GMFGen)
Develop mapping definition (GMFMap)
Edit generation parameters
Domain Model (ECore)
Generate diagram plug-ins
Run on GMF Runtime Platform

Figure 17. GMF development flow as a BPMN process

Creation of GMF-based graphical editor consists of the following steps:

- Creation of a GMF project and importing a domain model expressed in EMF;
- Creation of a graphical definition, that defines graphical elements to be displayed in the editor;
- Creation of a tooling definition model, which is used to specify elements such as pallet, tools, and actions for graphical elements;
- Creation of a mapping model, that defines mappings between the domain model elements and the graphical elements;
- Generation of a graphical editor (plug-in).

The result of the code generation is a fully functional graphical editor, in the form of an Eclipse plug-in.

2.2. Service compositions

As noted in the introduction, Web services are “self-describing, open components that support rapid, low-cost composition of distributed applications” [99], and they can be composed in entities that support automated execution of business processes (called service compositions). While a service-oriented architecture is a software architecture style focusing ”on how services are described and organized to support their dynamic automated discovery and use” [16].

Service compositions can generally be interpreted as implementation of business processes (single composition service) from services. Resulting service compositions can be (re)used in further service compositions. Such compositions can be offered as a complete application [99]. Currently, there are two well-known business interaction protocols that compose services, and they are known as “orchestrations” and “choreographies”.

A service orchestration is an interaction between services at a message level controlled by one party. According to the W3C’s Web service glossary [135]: “An orchestration defines the sequence and conditions in which one Web service invokes other Web services in order to realize some useful function. I.e., an orchestration is the pattern of interactions that a Web service agent must follow in order to achieve its goal.” Orchestration is the interaction between services at a message level controlled from one party. Such business processed can result in a “long-lived, transactional, multistep process model” [99]. There is presently a standard for service orchestrations called Web Services Business Process Execution Language (WS-BPEL) language [49] or for short BPEL. The current research demonstrates that the most common approach to modeling service orchestrations is based on the principles of business process modeling. In the context of orchestration modeling, BPMN [88] and UML activity diagrams are two typically approach. Actually, in the recent BPMN specification, there is a partial mapping defined between BPMN and BPEL [49]. We can consider this from the perspective that service compositions include creation of a business processes (single composition service) from composite Web services. Resulting service composition can be used in further service compositions

A service choreography, on the other hand, defines a message exchange that occurs between services. In choreography, every party defines their own part in the interaction. According to the Web service glossary, “a choreography defines the sequence and conditions under which multiple cooperating independent agents exchange messages in order to perform a task to achieve a goal state” [135]. This definition is further specialized in the Web Service Choreography Description Language (WS-CDL) candidature recommendation [55]: “a *choreography* defines re-usable common rules that govern the ordering of exchanged messages, and the provisioning patterns of collaborative behavior, as agreed upon between two or more interacting participants.” The key aspect is messages exchanged among collaborating parties, which agree on rules for ordering of messages. There are two approaches to modeling of choreographies [25]: interaction models and interconnected interface behavior models (interconnection models). Interaction models are built up of basic interactions (message exchanges), while interconnected interface behavior models define control flows of each participant a choreography. The representatives for the interaction model approach are WS-CDL [55], Let’s Dance [150] and iBPMN [25]. Interconnected interface behavior models can be represented in BPMN [88] and BPEL4Chor [23].

In the rest of the section, we give a short introduction to the Web services, and we describe two main service compositions languages, WS-BPEL for orchestrations and WS-CDL for choreographies.

2.2.1. Web services

There are three well-known definitions of Web services: “*A Web service is a loosely coupled software component that exposes functionality to a client over the Internet (or an intranet) by using web* *standards such as HTTP, XML, SOAP, WSDL, and UDDI*" [131]. The second definition: "*A Web Service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web Service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards*."[135]. The third definition "*A Web service is a software system identified by a URI, whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems. These systems may then interact with the Web service in a manner prescribed by its definition, using XML based messages conveyed by Internet protocols*"[3].

The Web services technology is based on three main specifications:

- SOAP is an XML-based protocol used for invoking operations and exchanging messages, in a distributed environment [134]. The SOAP protocol is based on SOAP messages that are exchanged among SOAP nodes, where every message has an optional header used for storing meta-information and a mandatory body. A commonly used transport protocol for SOAP is HTTP.
- WSDL (Web Services Description Language) is a language used for description of Web services, i.e., its interfaces and operations [136]. An operation is a point of interaction with a service, which consists of a message exchanged between the service and other parties. WSDL also describes an endpoint, which represents a point of contact for a service and its physical location.
- The UDDI (Universal Description, Discovery and Integration) protocol is used to discover and publish Web service descriptions in a central service registry [87].

In Figure 18, we show the Web service platform stack. It has five main layers: transport, messaging, description, quality of service and compositions. The Transport layer is used by the Messaging layer technologies to transport messages. The second layer (Messaging) is used for distributing messages between parties. The Description layer is a layer where WSDL is placed and it is used for describing Web services. The Quality of service layer deals with service cooperation, monitoring and service level agreements. In addition, on top is the Compositions layer where orchestration and choreography languages reside. All of these layers are supported by native compatibilities, such as publication and discovery of Web services, realized with UDDI.

[3] http://www.w3.org/TR/wsa-reqs/

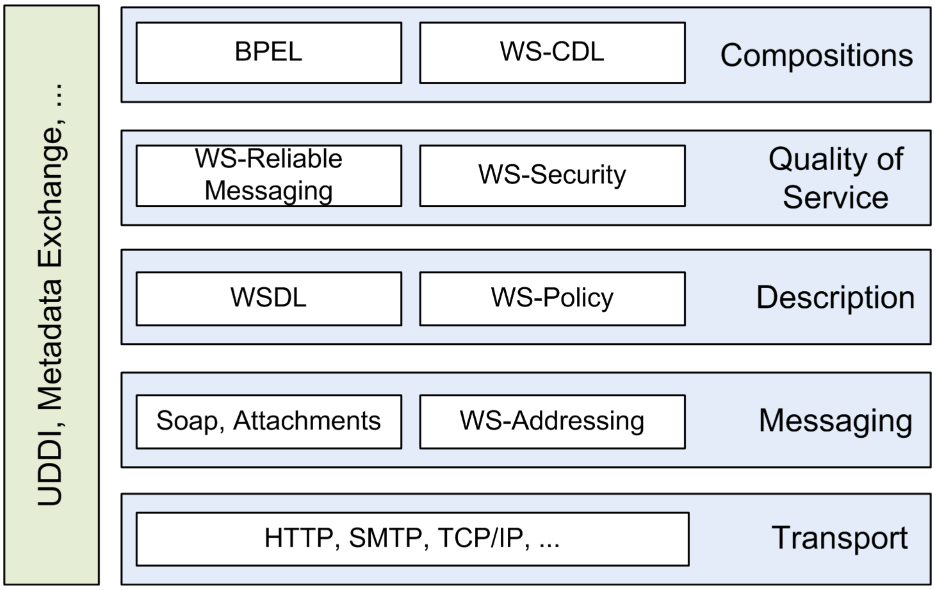


Figure 18. Web services platform (adapted from [145])

2.2.2. Process orchestrations - WS-BPEL

BPEL is a process-based language for defining interactions between partners [49]. It is XML-based language for defining service interactions between Web services participating in an interaction (partners). BPEL is developed by Microsoft, IBM, Siebel Systems, BEA, and SAP. The main result of a BPEL composition is a process and the basic BPEL element is an activity, which can be *basic* or *structured*. A partner involved in an interaction is identified by a WSDL interface and defined as a `partnerLink`. WSDL is also used in BPEL to define the public entry and exit points for a process and data types that are passed between partners. BPEL has a support for exception handling through the *throw* and *catch* clauses. BPEL compositions can be abstract or executable, where former capture non-executable interactions between services, the later is intended to be deployed on an execution engine.

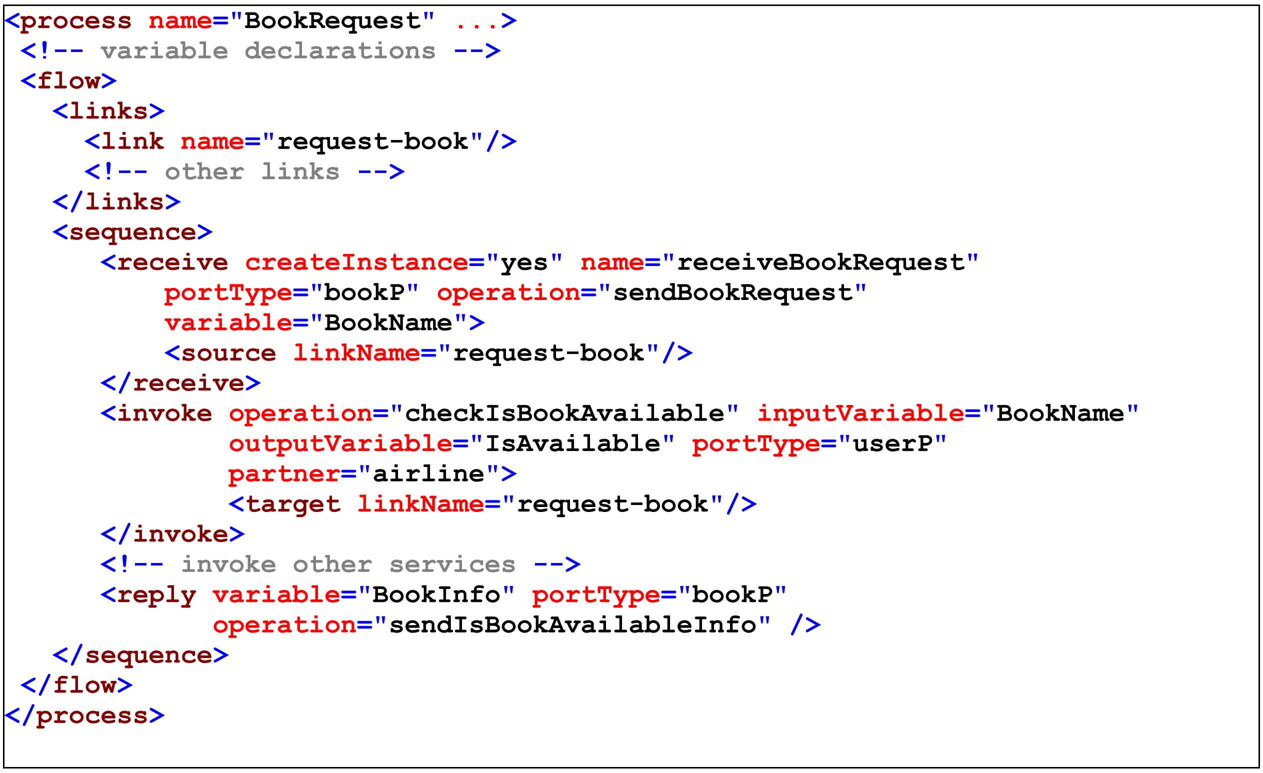
Basic BPEL elements are used for a simple communication:

- defining a process: `<process>`;
- invoking and receiving a partner service invocation: `<invoke>`, `<receive>`, `<reply>`;
- variable support: `<assign>`, `<empty>`, `<copy>`;
- defining partners in communication: `<partnerLink>`.

Structured BPEL elements are used for control flow and imperative programming:

- sequence and parallel flow: `<sequence>`, `<flow>`;
- loops and conditional branches: `<while>`, `<switch>`.

An example of a BPEL business process for a simple book loan request is given in Figure 19.



```
<process name="BookRequest" ...>
 <!-- variable declarations -->
 <flow>
   <links>
     <link name="request-book"/>
     <!-- other links -->
   </links>
   <sequence>
      <receive createInstance="yes" name="receiveBookRequest"
          portType="bookP" operation="sendBookRequest"
          variable="BookName">
          <source linkName="request-book"/>
      </receive>
      <invoke operation="checkIsBookAvailable" inputVariable="BookName"
              outputVariable="IsAvailable" portType="userP"
              partner="airline">
              <target linkName="request-book"/>
      </invoke>
      <!-- invoke other services -->
      <reply variable="BookInfo" portType="bookP"
             operation="sendIsBookAvailableInfo" />
   </sequence>
 </flow>
</process>
```

Figure 19. Book request BPEL process

A BPEL process is executed by a BPEL engine such as ActiveBPEL, IBM WebSphere or Microsoft BizTalk.

2.2.3. Process choreographies - WS-CDL

WS-CDL is a declarative XML-based language for describing message exchange and information handling by involved parties in order to achieve a business goal [55]. It is not designed as an execution language. The WS-CDL specification defines a choreography that can be seen as a contract that has been made between participants. Each participant has to conform to the contract by defining their own services. Each choreography description in WS-CDL consists of a WS-CDL document, which describes participants and their relations and optional interface of the participating services, defined by using WSDL or Java.

In Figure 20 we show main WS-CDL concepts and resulting XML elements. Every XML element is drawn as rectangle.

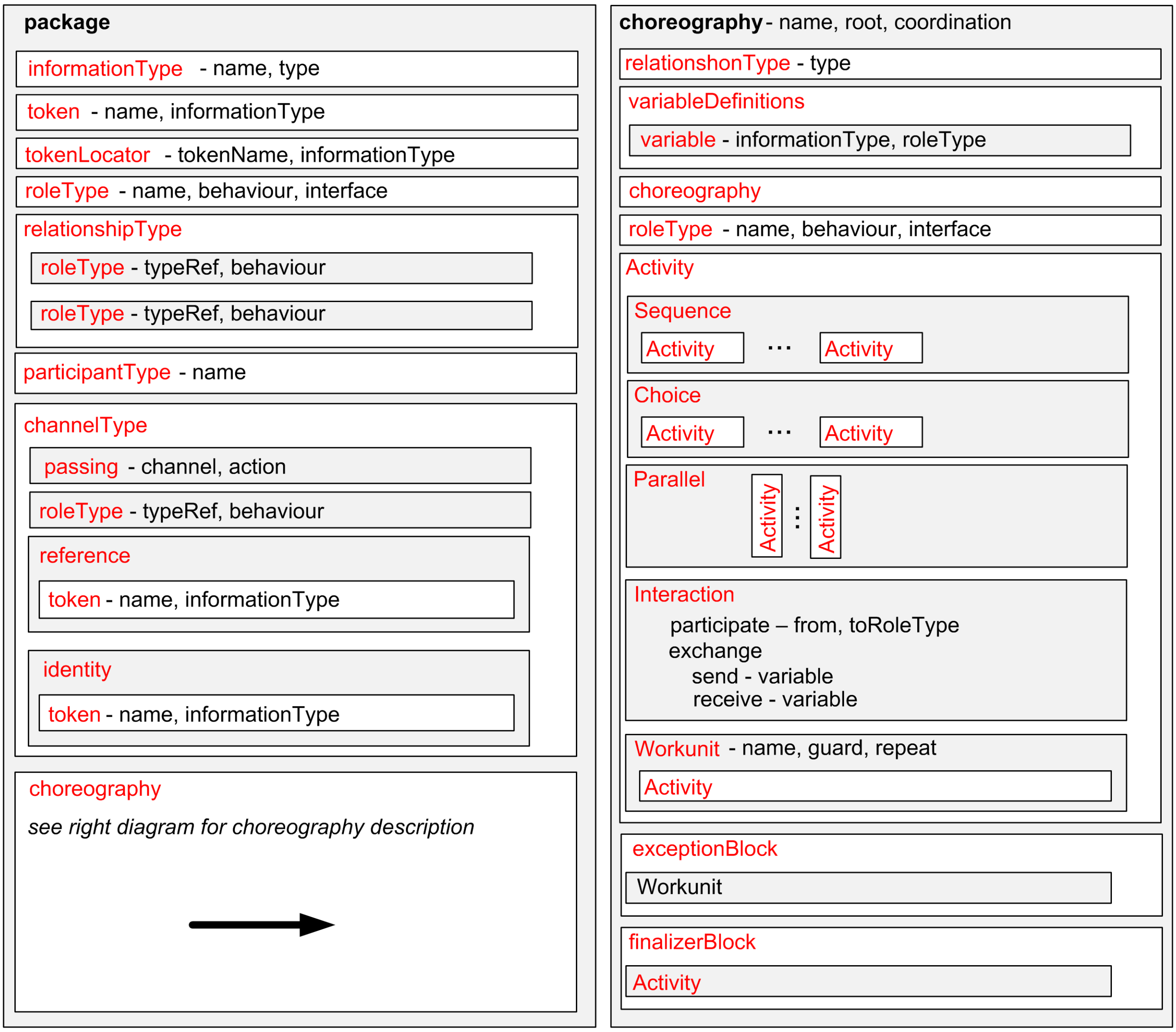

package
informationType - name, type
token - name, informationType
tokenLocator - tokenName, informationType
roleType - name, behaviour, interface
relationshipType
roleType - typeRef, behaviour
roleType - typeRef, behaviour
participantType - name
channelType
passing - channel, action
roleType - typeRef, behaviour
reference
token - name, informationType
identity
token - name, informationType
choreography
see right diagram for choreography description
choreography - name, root, coordination
relationshonType - type
variableDefinitions
variable - informationType, roleType
choreography
roleType - name, behaviour, interface
Activity
Sequence
Activity ... Activity
Choice
Activity ... Activity
Parallel
Activity ... Activity
Interaction
participate – from, toRoleType
exchange
send - variable
receive - variable
Workunit - name, guard, repeat
Activity
exceptionBlock
Workunit
finalizerBlock
Activity

Figure 20. WS-CDL structure [13]

Information handling. WS-CDL types are modeled by using *informationTypes*, and they reference type definitions defined as a WSDL or XML Schema elements. They are referenced by *variables* and *tokens*, where tokens reference to an *informationType* and *informationTypes* provide identification for *channelTypes*. Variables contain information exchanged between of the *roleTypes*.

Interactions. In WS-CDL, exchanged information is modeled between participants as seen from a global viewpoint . Every participant is a requester of a service, but in the same time a provider of another service. Participant is modeled by the *participantType*, and it plays a set of roles, so it contains one or more *roleTypes*. A *roleType* represents one on several observable behaviors of a participant through message exchanges (identifies a WSDL interface type). A *relationshipType* contains exactly two roles (*roleTypes*), which interact in a choreography. A *channelType* specifies where and how information is exchanged between participants. It references a *roleType* which is a target of an information exchange.

Activities. A choreography can include one of three types of activities: basic activities, ordering structures and WorkUnit activities. Basic activities define interactions on the choreography flow and basic activity can be one of the following types:

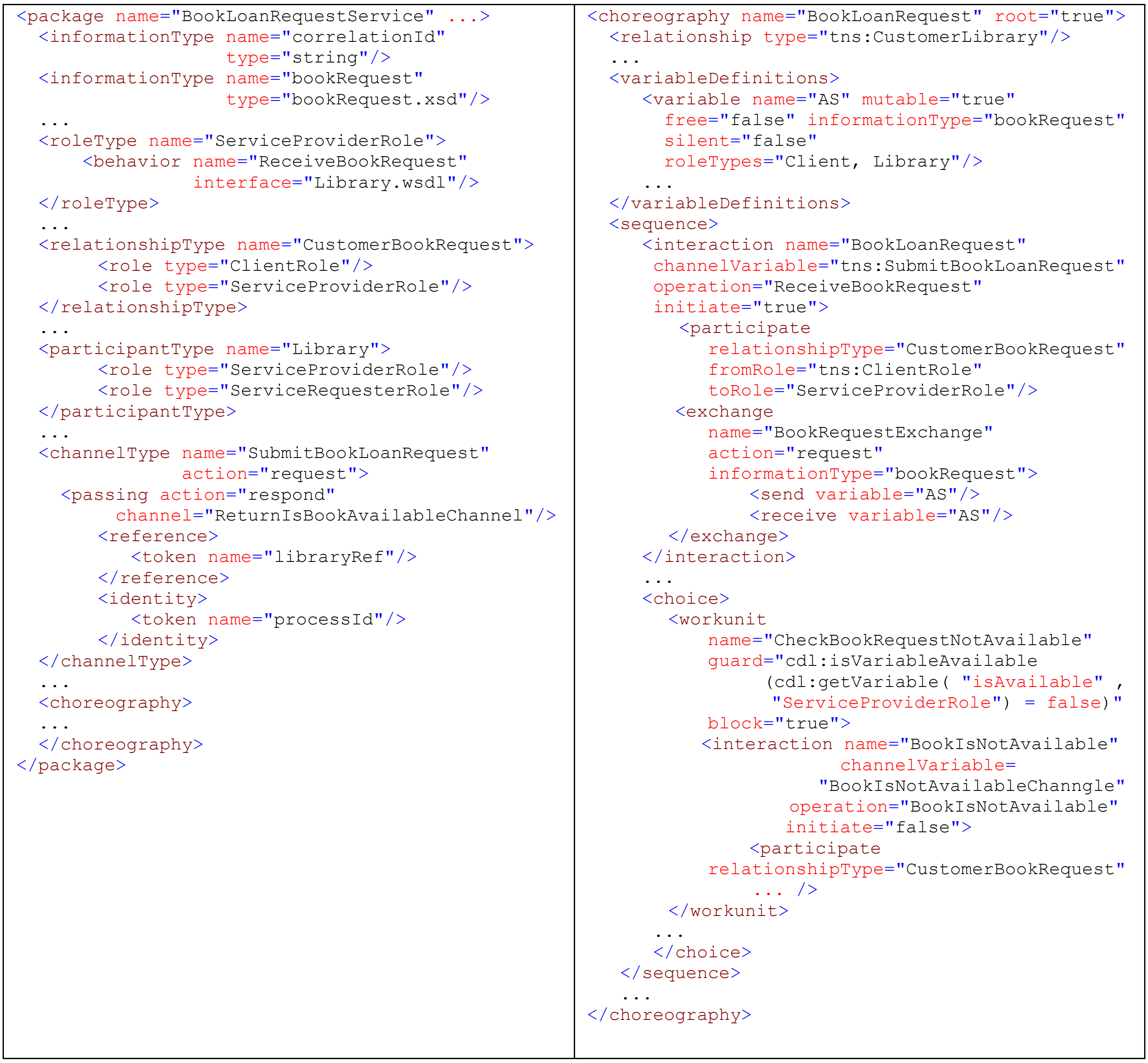
- *Interaction activity* describes what information to be exchanged among participants. It has a *channelVariable*, which binds to a *channelType* and a WSDL interface, and a SOAP operation, which is defined through the WSDL interface. This activity also includes a *participate* element that defines receiving and sending role, and an *exchange* element which variables are used in an interaction and is action request or response type.

- *Assign activity* is used for creation and manipulation of variables.
- *SilentAction* activity defines where one or all participants in the choreography perform actions with non-observable behavior.
- *PerformActivity* is used to perform a separate choreography.
- *NoAction* is used to define where a participant does not perform any action.
- *FinalizeActivity* concludes a choreography.

Ordering structures are used to specify a control flow by ordering activities. Those structures are: *sequence,* for handling activities in a sequential order, *parallel* for handling activities in a parallel order, and choice for handling Data-driven (based on a condition) or Event-driven activities (hold until an event occurs or while a variable is populated).

WorkUnits are used to group one or more activities into a single unit with a conditional execution of such activites. The condition can be *repetitive* (*repeat* is set to true and enclosed activities are repeated upon being completed), *competitive* (multiple workunits are defined inside choice) or *blocking*. The conditional statement is defined by the *guard* condition, which determines whetether workunit is performed at all. If guard condition is false then workunit is skipped.

In Figure 21, we show an example of a simple book loan request choreography. In Figure 21a, we show *package information*, where a package element is the root element of the choreography. The package element contains *informationType* definition and variables (*bookRequest*). A *roleType* represents an actor in a message exchange and it associaties operation name, as well as its WSDL interface by using the behavior element. In this example, *ServiceProviderRole* implements the WSDL *ReceiveBookRequest* operation. The *relationShip* type connects two roles, i.e., *CustomerBookRequest* associates the *ClientRole* to the *ServiceProviderRole*. The *participantType* groups two roles, and the *channelType* defines the role that the receiver of a message plays, that is, a return channel for the response to a submission. Every package contains choreography definition that specifies relationships. In Figure 21b), there is a relationship between the *Customer* and the *Library*. Next, variables are declared, that are used by the *ClientRole* and the *ServiceProviderRole*. The interaction element is used to define a communication and direction of a message from a sender (*fromRole*) to a receiver (*toRole*). The exchange element contains name of the operation used in interaction. Then, *WorkUnit* is defined by using the WS-CDL functions *isVariableAvailable* and *getVariable* to get variable information from a Library and based on that a notification of book availability is sent to the Client.



```
<package name="BookLoanRequestService" ...>
  <informationType name="correlationId"
                   type="string"/>
  <informationType name="bookRequest"
                   type="bookRequest.xsd"/>
  ...
  <roleType name="ServiceProviderRole">
      <behavior name="ReceiveBookRequest"
                interface="Library.wsdl"/>
  </roleType>
  ...
  <relationshipType name="CustomerBookRequest">
       <role type="ClientRole"/>
       <role type="ServiceProviderRole"/>
  </relationshipType>
  ...
  <participantType name="Library">
       <role type="ServiceProviderRole"/>
       <role type="ServiceRequesterRole"/>
  </participantType>
  ...
  <channelType name="SubmitBookLoanRequest"
               action="request">
    <passing action="respond"
         channel="ReturnIsBookAvailableChannel"/>
       <reference>
          <token name="libraryRef"/>
       </reference>
       <identity>
          <token name="processId"/>
       </identity>
  </channelType>
  ...
  <choreography>
  ...
  </choreography>
</package>
```

```
<choreography name="BookLoanRequest" root="true">
  <relationship type="tns:CustomerLibrary"/>
  ...
  <variableDefinitions>
     <variable name="AS" mutable="true"
       free="false" informationType="bookRequest"
       silent="false"
       roleTypes="Client, Library"/>
     ...
  </variableDefinitions>
  <sequence>
     <interaction name="BookLoanRequest"
      channelVariable="tns:SubmitBookLoanRequest"
      operation="ReceiveBookRequest"
      initiate="true">
        <participate
           relationshipType="CustomerBookRequest"
           fromRole="tns:ClientRole"
           toRole="ServiceProviderRole"/>
        <exchange
           name="BookRequestExchange"
           action="request"
           informationType="bookRequest">
              <send variable="AS"/>
              <receive variable="AS"/>
        </exchange>
     </interaction>
     ...
     <choice>
        <workunit
           name="CheckBookRequestNotAvailable"
           guard="cdl:isVariableAvailable
                (cdl:getVariable( "isAvailable" ,
                 "ServiceProviderRole") = false)"
           block="true">
          <interaction name="BookIsNotAvailable"
                channelVariable=
                  "BookIsNotAvailableChanngle"
               operation="BookIsNotAvailable"
               initiate="false">
              <participate
           relationshipType="CustomerBookRequest"
               ... />
        </workunit>
         ...
       </choice>
   </sequence>
   ...
</choreography>
```

a) b)

Figure 21. WS-CDL choreography example

Choreography modeling in WS-CDL is supported through a tool called Pi4soa [103].

2.2.4. Process choreographies – BPEL4Chor

BPEL4Chor is created by Decker et al. [23] [24] [25] and represents an extension of a BPEL language for modeling choreographies, i.e., interconnected interface behavior descriptions for defining choreographies. BPEL4Chor is built in the way that it abstract communication activities of choreographies (i.e., elementary interactions) from technical configuration.

Modeling a choreography in BPEL4Chor includes following activities:

1. Definition of the participants and participant types.
2. Definition of the message links between participants, i.e., data artefacts exchanged between participants.
3. Specification of the behavioral depedencies and data flow between message exchanges, i.e., order of messages.

4. Connection (grounding) of an each message link to the concrete services. This includes defining of serialization formats for the messages, etc., and it represents an optional activity.

BPEL4Chor also includes a three different artefact types:

1. Partticipant behavior description, defines the control flow depedencies between activites.
2. Participant topology is used to define structural aspects of a choreography, and this includes specifying participant references, types and message links.
3. Participant grounding define the concrete technical configuration for a choreography, i.e., links to WSDL and XSD definitions.

In addition, it is possible to create an executable BPEL process from the BPEL4Chor participant behavior description, by defining the parcitipant grounding for each participant. This represents an input for the transformation which transforms such an description to the executable BPEL process [24].

2.3. Business processes

In this section, we introduce key concepts of business processes, as well as languages for representing business process models.

2.3.1. Concepts and terminology

According to Weske [146], a business process consists "of a set of activities that are performed in coordination in an organizational and technical environment. These activities jointly realize a business goal. Each business process is enacted by a single organization, but it may interact with business processes performed by other organizations". A business process can also be defined as "a process that creates a value or result for a customer. It is directed by the business objectives of a company and by the business environment" [107]. Business process consists of activities that are executed in coordinated way in order to achieve some goal and they are represented with business process models. According to [146], activities can be system activities (activities which does not involve human user, they are entirely executed by software), user-interaction activities (activates that workers perform using software such as entering data on a form), or manual activities (that are not supported by software). Workflows realize a part of a business process, so it is associated with a process. Business process modeling is an activity that includes different concepts, such as business process, workflow and activity.

Business process management includes "concepts, methods, and techniques to support the design, administration, configuration, enactment, and analysis of business processes" [146]. Business process management complexity is distributed through different levels of abstractions [146]. There is horizontal abstraction that denotes separation of modeling levels, from the instance and the model level to the (meta) metamodel level. The horizontal abstraction separate concepts as identified by the OMG in the MDA [85]. This abstraction is shown on MOF in Section 2.1.5.1. Along with horizontal abstraction, often it is needed to separate subdomains in order to integrate modeling efforts in different subdomains, such as: functional modeling, information modeling, organization modeling and IT landscape modeling.

In order to define basic concepts of business process models from metamodelling point of view, different concepts are identified [146]:

- *Process model* is a collection of different activities used to produce certain goals. A process models contains related nodes and directed edges.
- *Edge* is used to express connections between nodes in a process model.
- *Node* represents activity model, event model or a gateway model in a process model. An activity model is a unit of work in a process model that can appear only once and activity model is usually represented with rectangle with rounded edges. Every activity model

"can have exactly one incoming edge and exactly one outgoing edge" [146]. An event model represents a state occurrence in a process model, such as start or end event models. Such models are usually represented with circles in business process modeling languages. A gateway model represents control flow constructs, such as: split, join nodes and sequences. Such models are usually represented with diamonds.

Along with basic process model concepts, there are two more important issues: process instances and process interactions [146]. A process instance is always connected to one process model and it contains an arbitrary number of activity, event or gateway instances, all of which are associated with exactly one process model. Process interactions are interactions between different parties. These interactions are usually defined in a peer-to-peer way, by following predefined choreography or orchestration [146].

2.3.2. Business Process flexibility and variability

Flexibility is one of the important properties of the business processes, and flexibility in this case means that business process should be easily adapted to the frequent changes. Regev et al. [109] defines business process flexibility as "*the capability to implement changes in the business process type and instances by changing only those parts that need to be changed and keeping other parts stable*". However, it is hard to measure such flexibility. *Kasi and Tang* [53] proposed a framework for comparison of business processes, where they described flexibility of business processes in the three dimensions:

- Time – the process should adapt to change more quickly;
- Cost – the process should adapt to change with less cost;
- Ease – the process should adapt to change with maximum ease.

From this, we can conclude that process flexibility could be achieved with less both time and costs; for the ease dimension, we can say that is important to change a minimal number of elements in minimal number of places.

Along with flexibility, one important concept in SOA and business process modeling is variability, which is defined as a "property of an object of being changeable" [48]. It has been reported recently that there has been paid little attention to variability of business processes [48]. However, in [119] authors identified different variability mechanism that can be used in process models. They addressed some basic variability mechanisms, such as: encapsulation of varying sub-processes, parameterization, addition, omission, and replacement of single elements, and data type variability. These variability points are shown in BPMN diagrams and implemented with Java variability mechanisms and code generators. *Goldszmidt & Osipov* [41] propose a more general solution to implementing variability in workflows, called "point-of-variability". They define point-of-variability as "locations where decisions are implemented that are likely to change and thus should be externalized". By using the points-of-variability it is possible to choose among more alternatives for one workflow element (such as activity) based on some predefined condition. Business rules are presented as one of the possible implementations for variability points at runtime. *Ejindhoven et. al* [30] followed this approach in their proposed methodology, where variability points are identified in the first step of the methodology. Later, those variability points are modeled by means of workflow patterns, which can use business rules in order to be more expressive.

2.3.3. Business Process Modeling Languages (BPML)

In this section, we describe main business process modeling languages, including Event Driven Process Chains, Petri Nets, Integrated DEFinition Method 3 (IDEF3), UML 2.0 Activity Diagrams, Agent Object-Relationship Modeling Language (AORML), Lets' Dance, Business Process Modeling Notation (BPMN) and iBPMN. We chosed these languages because they represent a de facto standard in academics and industry in the modeling of business processes.

2.3.3.1. Petri nets

Petri nets are designed for modeling, simulation and analysis of processes and systems [100] and they can be used for specifying business processes in a formal way. Carl Adam Petri introduced Petri Nets in his dissertation [102]. Originally, Petri Nets were designed to model dynamic systems with static structure by using a graphical syntax. There are different types of Petri Nets, such as colored Petri Nets [52], relation Petri Nets [7], condition/event nets, place/transition nets and predicate/transition nets.

A Petri Net is directed graph that consist of places, transitions, tokens and directed arcs. Transitions have input and output places and they are interpreted as actions, activities or events that cause a change of a state. Places represent possible states of the system. In the graphical notation, places are represented with circles, connectors by directed arcs and transitions by rectangles. Tokens are placed within a state, which is called marking, and the state of a system is recorded through different token states. Transitions have input and output places, where input places of transition are the places at the sources of its incoming arcs, and transition's output places are places to which arcs run from a transition. A transition may fire if it is enabled. It is there is a token in each of its input places. When a transition fires, the transition uses tokens from its input places and performs a task. Then, the transition places a specified number of tokens into each of its output places. In addition, multiple transitions can be enabled at the same time.

A simple Petri Net representing a process model is shown in Figure 22. This Petri Net has two places (p1 and p2) and one transition (t1). When the first place fires, the second place receives a token. The process starts when the token is placed on place p1, and the token is represented by the black dot in that place.

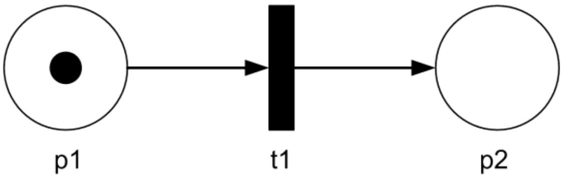


Figure 22. Simple Petri Net

One important type of Petri Nets is colored Petri Nets. In standard Petri Nets, it is impossible to distinguish types tokens, and this shortcoming in is addressed by the color feature, where every token has a value. A Petri Net metamodel proposal is shown in Figure 4 and in [36].

2.3.3.2. Event Driven Process Chain (EPC)

Event-Driven Process Chain (EPC) [117] is a graphical business modeling language developed by August-Wilhelm Scheer in 1992 at the University of Saarland in Germany in collaboration with SAP AG, as part of the ARIS framework (Architecture of Integrated Information System). EPC was developed with the goal to be understood by business people. The EPC approach is usually denoted with a ARIS house with three pillars and a roof, as shown in Figure 23. The roof represents an organizational view, while pillars represent data, control and functional view. The focus of control view is on integration of all other views and it provides links among the artifacts in those views. It integrates all elements that are designed separately in different views, into a common context and a resulting model is EPC.

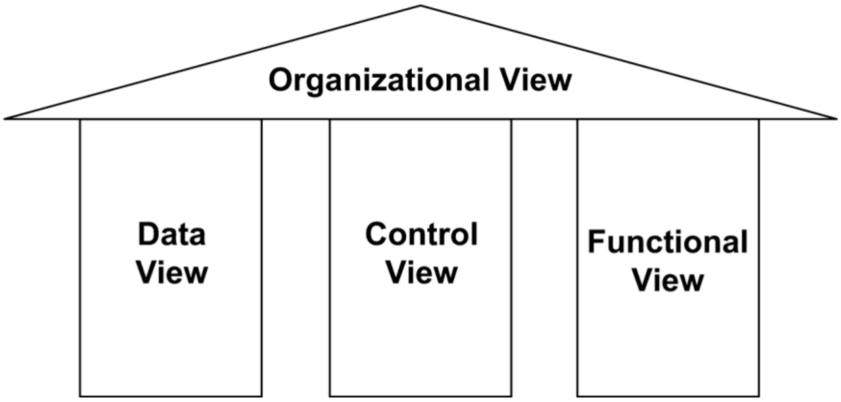


Figure 23. ARIS business process framework [117]

The *data view* is expressed by using the Entity Relationship (ER) diagrams or by using the UML class diagrams and it consists of events and statuses. This view contains different data objects that are used by function during process execution. The *functional view* contains descriptions of relationships between functions, description of sub-goal and goals that need to be performed. The *organizational view* includes relationships between organizational units of an enterprise at the type and instance levels, as well as organizational aspects of information technology of the enterprise.

The main building elements of Event-Driven Process Chains are shown in Figure 24.

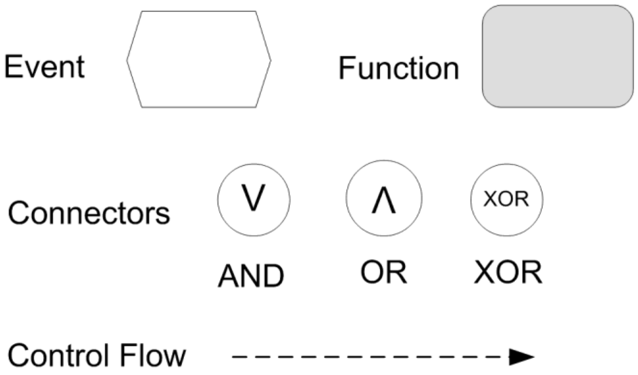


Figure 24. Notation of EPC elements

Those elements are:

- *Functions* in EPC model activities and tasks within the company and they represent units of work;
- *Events* are passive elements as they don not provide decisions. Events are created by functions or by actors outside of a model and they describe under what conditions a function works or in which state a function results.
- *Connectors* are used in EPC to connect functions and events in the control flow. The three types of connector exist in EPC: logical AND, OR, and exclusive or (XOR). They serve all as split and as join nodes.

We show simple an ECP diagram in Figure 25. This process start by receiving a request and this request is represented by an event, as an event EPC diagram must start with an event. After receiving the request, the request is analyzed and is either accepted or rejected.

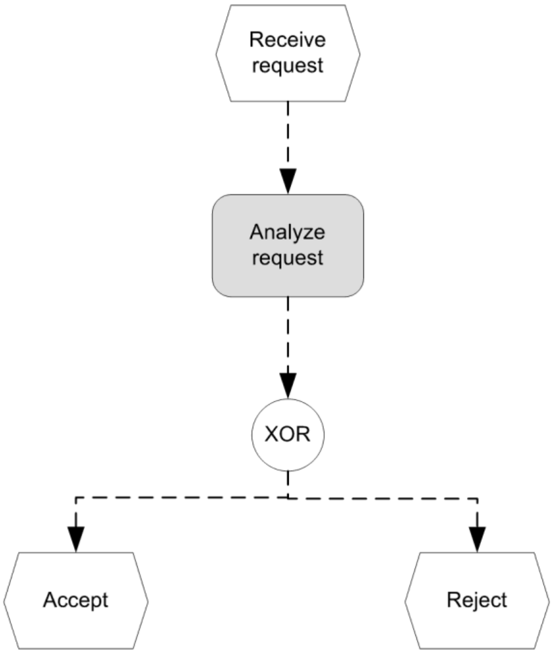


Figure 25. An example of Event-Driven Process Chain

There are currently two well-known metamodel proposals for EPC, one shown as a UML class model [63], and another shown as an ER model [120]. We here show the first metamodel, as it follows MDE metamodeling standard (see Figure 26). This metamodel support all EPC concepts, such as functions, events, logical operators and control flows.

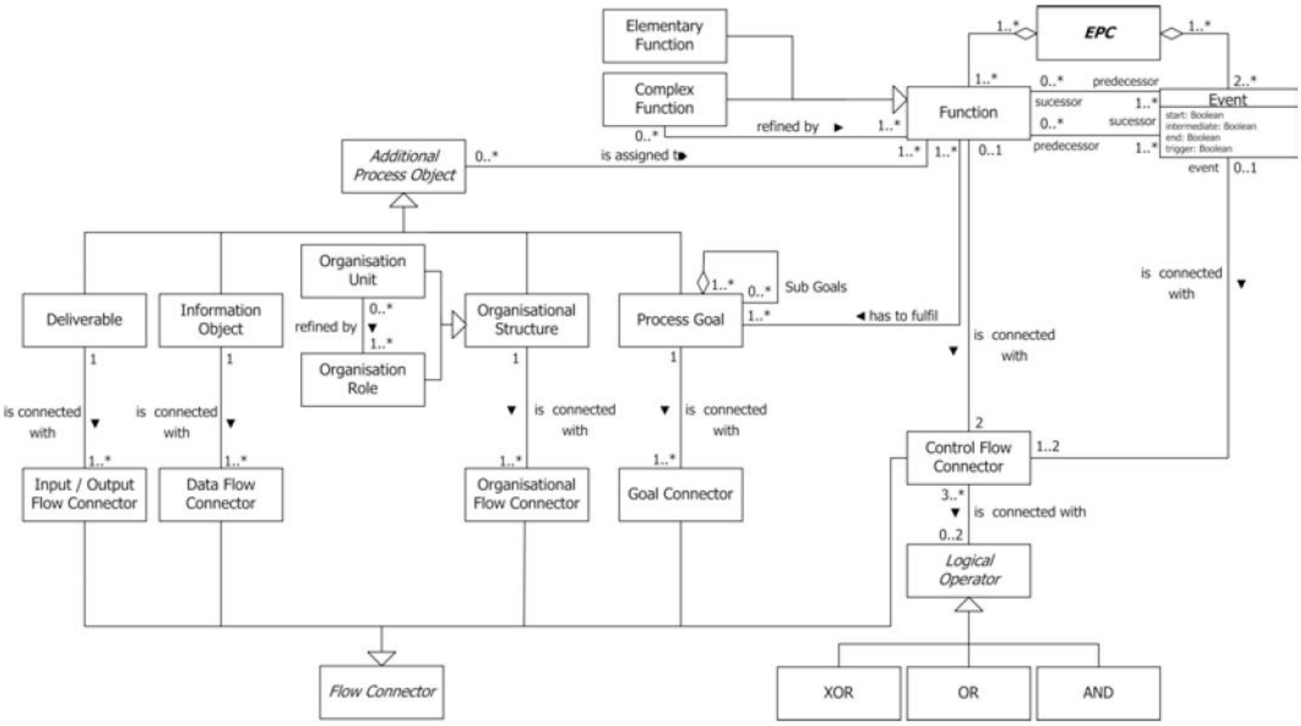


Figure 26. EPC metamodel [63]

2.3.3.3. Integrated DEFinition Method 3 (IDEF3)

IDEF3 is designed to model business processes and sequences of a system [75]. It provides the two perspectives: the process view (process sequence model) and the object view (objects-state-change model). The process view models a process sequence, while the object view describes object states which an object can have throughout a process. These two views contain units of information that form a system description, and they are called basic units.

IDEF3 process-centered strategy provides a visualization of process-centered descriptions of a scenario. It mainly consists of *Units of Behavior (UOB)*, *Links* and *Junctions*. IDEF3 also has references and notes, which are elements used across process and object schematics. These main elements are shown in Figure 27. UOB's represent activities in a business process. A UOB is graphically represented by a rectangle with a reference and label. If a UOB is complex, it is possible to decompose it into its components. *Link* describes relationship between UOB's and is represented with an arrow. *Junctions* are used to represent branches, and that is logical operators AND, OR and XOR.

IDEF3 support three types of links: *simple precedence links*, *constraint precedence links* and *dashed links*. Simple precedence links are used to express temporal precedence relations between UOBs. They are represented graphically by using a solid arrow and are most widely used link. Constraint precedence links add additional semantics to simple precedence links that every instance of source UOB must be followed with instance of destination UOB. Dashed links have no predefined semantics, so they are usually called relational links. This type of links shows existence of relationship between two UOB's.

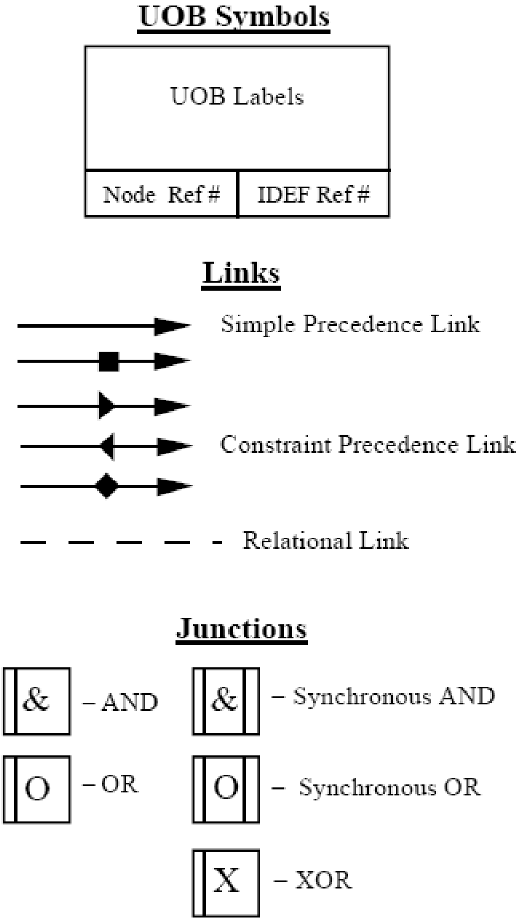


Figure 27. Symbols used for IDEF3 Process Description Schematics [75]

There are four types of *junctions* in IDEF3, two fan-out junctions: AND junctions are divergence branch points that involve multiple parallel subprocesses, while OR junctions are divergence branch points which involve different alternative subprocesses, and two fan-in junctions: AND and OR which represent points of convergence, involving multiple parallel subprocesses and multiple alternative subprocesses, respectively.

IDEF3 support four types of referents: UOB, Scenario, Transition Schematic and Go-to referents. A UOB referent type specifies that another instance of previously defined UOB occurs at a specific point in the process. A Scenario referent type indicates that next happening in the process flow is an occurrence of an activation of the referenced Scenario. A Transition Schematic referent type has to be initiated during an activation of its associated UOB. They are connected through a simple connecting link. A Go-to referent type references another UOB, and it is often used to describe loops in a process.

The Object View consists of Object States, Links, Relations and Junctions. These elements are shown in Figure 28. An object is of a certain *kind*, is represented simply by a circle and denoted by a label. Relations describe taxonomy relationships between objects, while transitions describe change from an Object A to an Object B, which are connected through Links. If a stronger connection between two objects is needed to be shown, a double-headed arrow is used.

In IDEF3 Individuals are referred as first-order objects, while second-order objects are Properties and relations that hold among individuals.

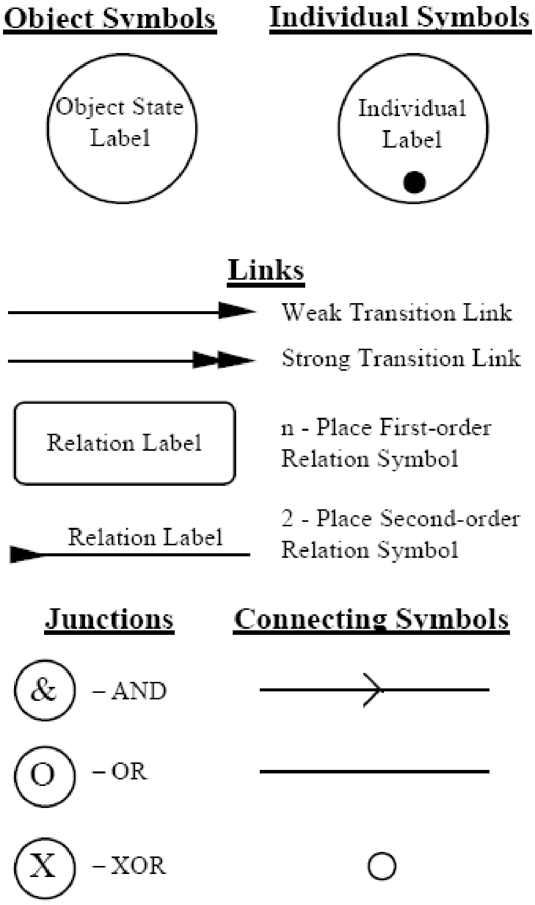


Figure 28. Symbols used for IDEF3 Object Description Schematics [75]

An example of a simple book request business process in the IDEF3 notation is shown in Figure 29.

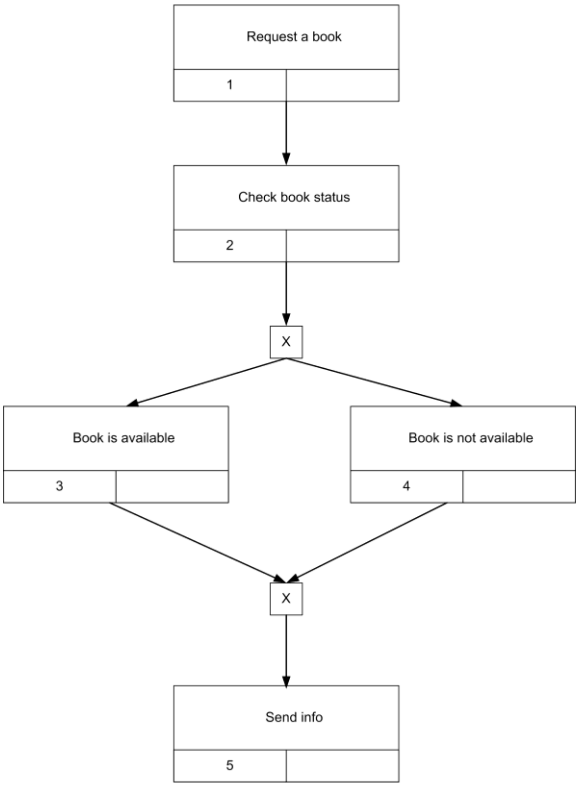


Figure 29. Example business process in the IDEF3 notation

2.3.3.4. UML 2.0 Activity Diagrams (AD)

Activity diagrams are one type of the seven behavioral diagrams in the UML 2.0 language that show sequence of action executions [96]. They are used to model actions (i.e., flow from activity to activity) and a business processes. Main elements of activity diagrams are actions nodes and activity partitions, where activity partitions are used to group action nodes. In Figure 30, we show main elements of UML 2.0 activity diagrams.

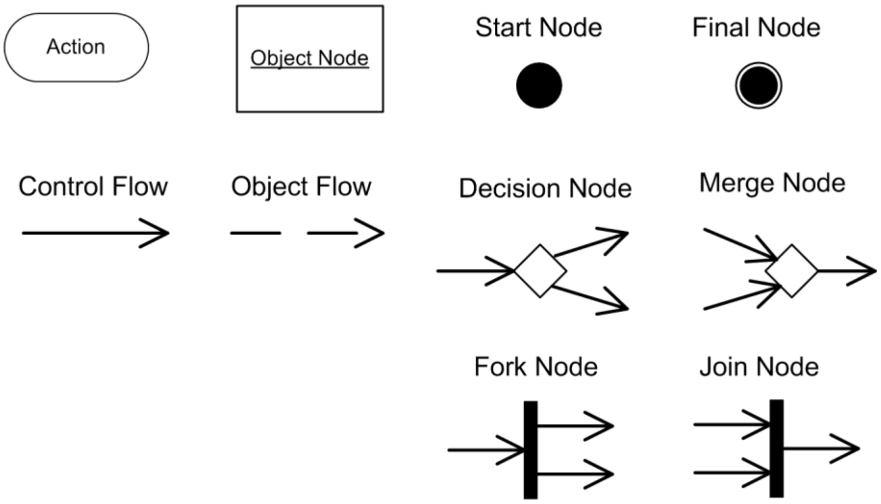


Figure 30. Basic elements of UML 2.0 Activity Diagrams

Actions are shown as rounded rectangles, with their name (verb) in them. Every activity diagram starts with the start node (initial node) and ends with the final node (activity final). A flow of activity among activity diagram elements is marked by using a line with an arrowhead. The Object Flow shows an object or a data passing between activities. The Merge node is used to synchronize multiple incoming flows into a single flow. The Join node is similar to the merge node with a difference that the join synchronizes two incoming flows and produces a single outgoing flow, and outgoing flow from a join cannot execute until all the incoming flows are received. The Merge passes control flow whenever it is reached by an incoming flow. The Decision accepts one incoming flow and can have multiple outgoing flows with conditions defined on each outgoing flow. Those conditions are represented in square brackets. The Fork node is used to split one incoming control flow into multiple outgoing flows.

An example of simple UML 2.0 Activity diagram for book request is shown in Figure 31.

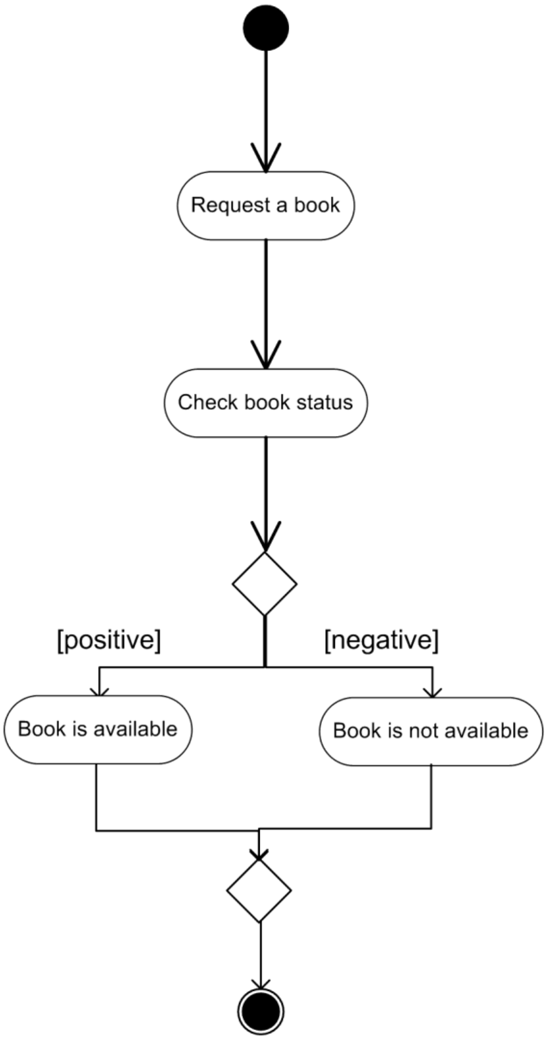


Figure 31. An example of UML 2.0 Activity diagram

UML 2.0 activity diagram elements are represented in UML 2 metamodel. Figure 32 shows an excerpt of the UML 2 metamodel for Activity Diagrams. This metamodel contains all elements of Activity diagrams, such as Activity, Action, ActivityPartition, ControlFlow, ObjectFlow, ForkNode, JoinNode, DecisionNode and MergeNode.

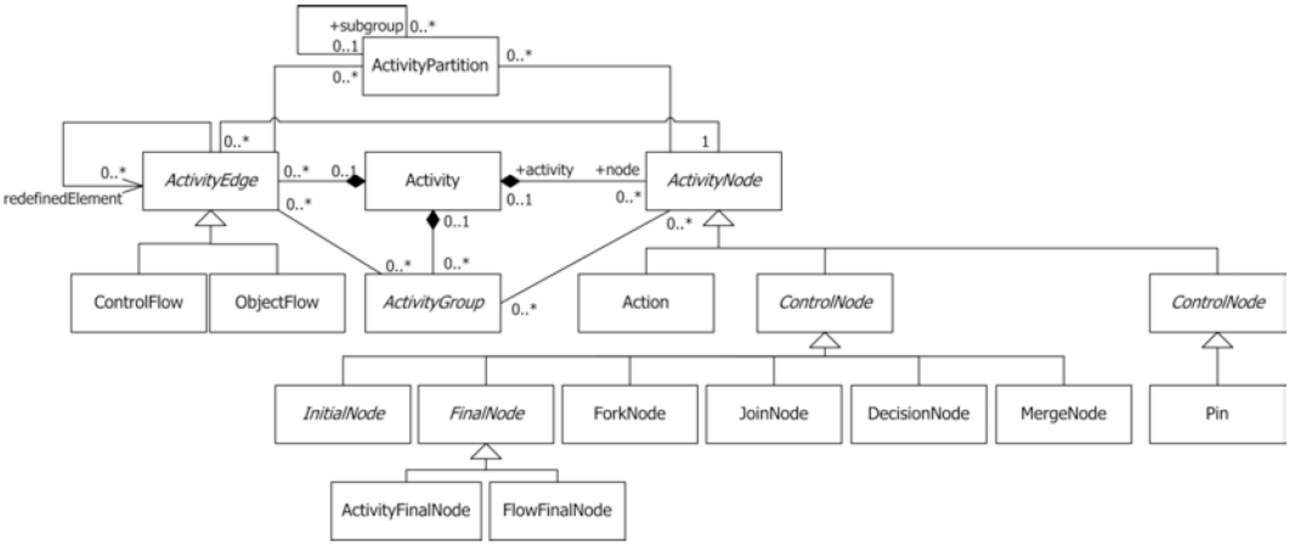


Figure 32. An excerpt of the UML 2 metamodel for Activity Diagrams [63]

2.3.3.5. Agent Object-Relationship Modeling Language (AORML)

The AOR Modeling language is a language for modeling organizational information systems [128] [129]. This language model business processes by modeling agents, events, actions, claims and commitments, with basic relationships from UML class and ER modeling, such as aggregation, association, and generalization. The AOR Modeling language follows the business agents-based approach, which includes six perspectives of agent-oriented modeling: organizational, informational, interactional, functional, motivational and behavioral. It can model all of these perspectives and to represent multiple perspectives on the same diagram by using a combination of goal-based use cases. Business processes are defined by modeling agent’s behavior, primarily by means of interaction patterns expressed in the form of reaction rules.

AORML agents can communicate, perceive, act, make commitments and satisfy claims, while objects are passive entities that do not have such capabilities. There are two types of AOR models, namely, *internal* and *external* models. An internal AOR model employs the first person view of a particular agent to be modeled, and an external AOR model shows a perspective of an external observer who is looking at agents and their interactions.

Figure 33 show basic elements of external AOR structure modeling. The AOR graphical modeling notation follows the UML 2.0 principles. The main elements of the external AOR structure modeling include agent types and instances, with their internal agent types and instances, their beliefs about objects and external agents, as well as relationships between agents [129].

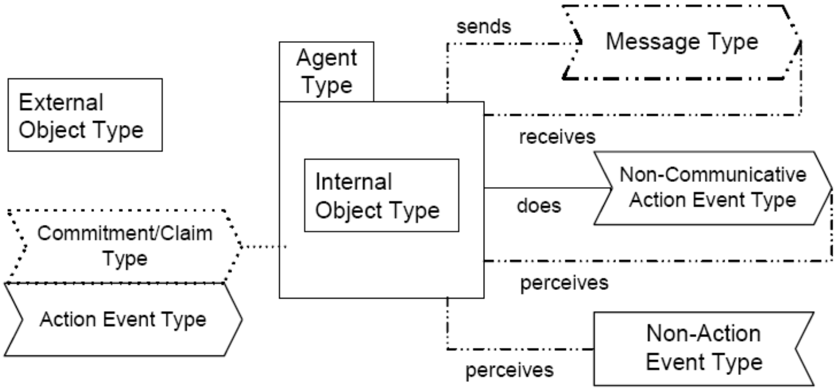


Figure 33. Basic elements of external AOR structure modeling [129]

The AORML graphical notation includes action event and non-action event types, and communicative action event and non-communicative action event type, as well as commitment/claim type, which are coupled with the action event type whose instances fulfill corresponding commitment. One of the main behavior modeling elements of AORML are reaction rules. They are used to express interaction patterns. Figure 34 contains an example of a AORML diagram with a reaction rule expressed by a circle with incoming and outgoing arrows, which are drawn within the agent rectangle. Every reaction rule has exactly one incoming arrow that specifies the triggering event type and two kinds of outgoing arrows: one for specifying changing beliefs and commitments and another one for specifying the performance actions. The outgoing connector with a double arrowhead denotes a mental effect, while the outgoing connector to an action event type denotes the performance of an action of that type [129].

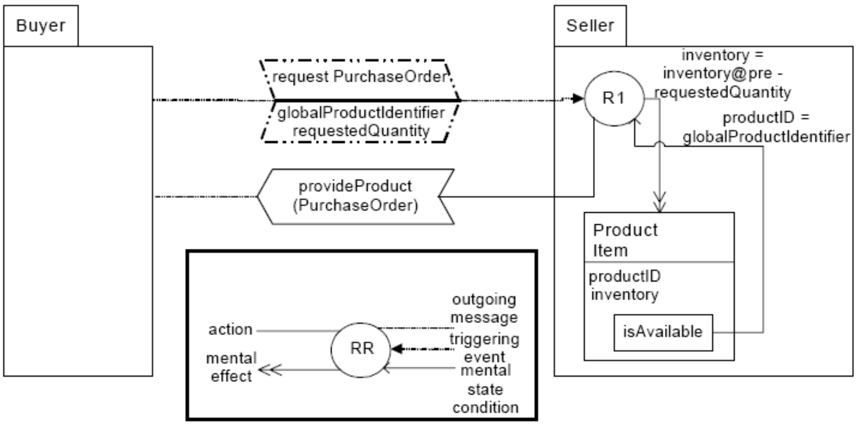


Figure 34. An example of AORML diagram [129]

Figure 34 shows an example of an AORML model where the Seller's reaction rule is used to perceive a communicative action event of the type *requestPurchaseOrder*. This rule has a precondition, which defines the availability of the *ProductItem* for the *Buyer* by checking the database of the *Seller*, and a post-condition, which affects the representation of the corresponding *ProductItem* in the *Seller's* database by decreasing its *inventory* attribute by the *requestedQuantity* value of the message received from the *Buyer*. Both, the precondition and post-condition are represented by using the Object Constraint Language (OCL) expressions [94].

The metamodel of AORML language is shown in Figure 35. It includes all views and concepts of agent-oriented modeling approach.

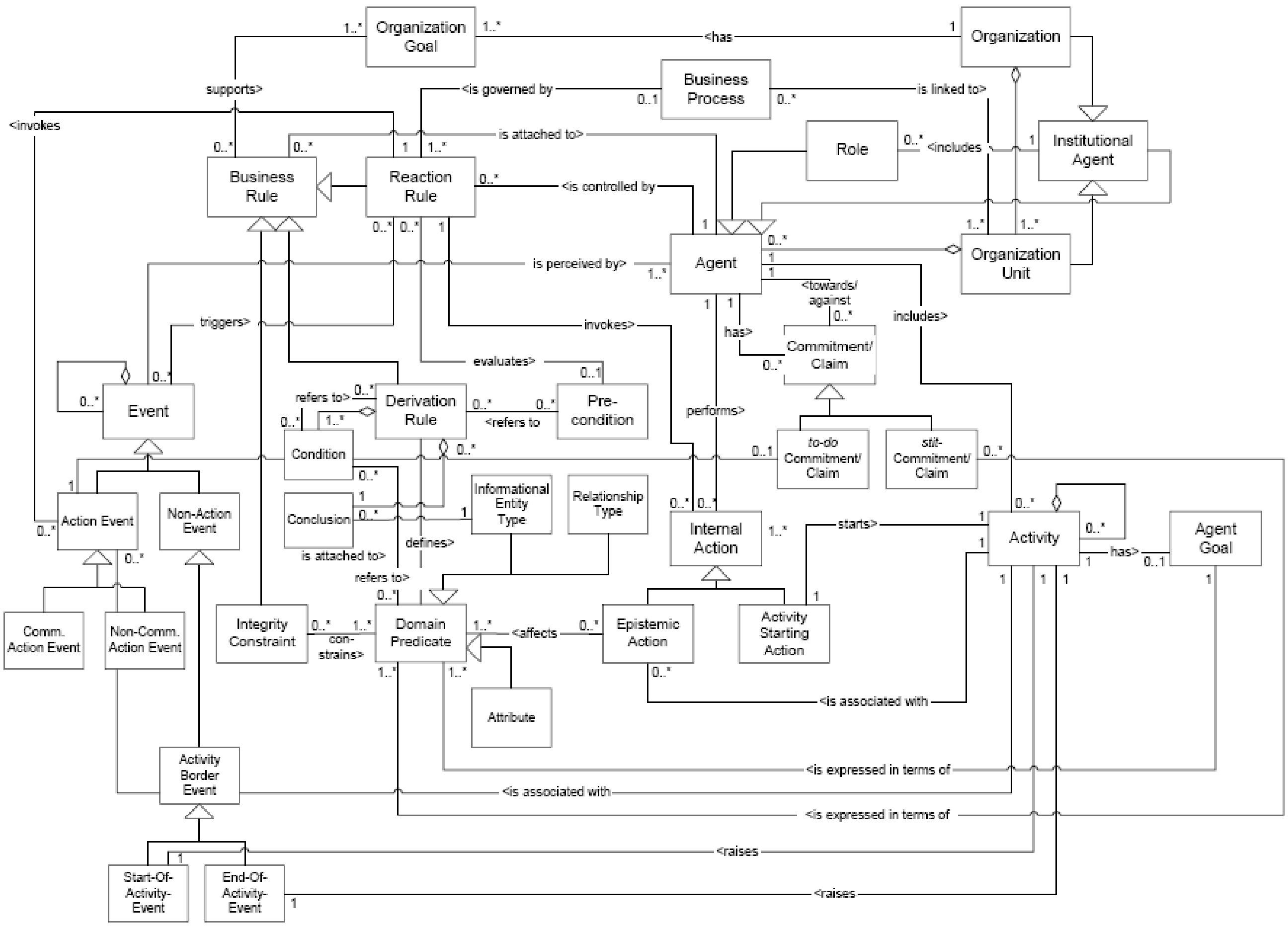


Figure 35. The metamodel of the AORML language [128]

2.3.3.6. Let's Dance

Let's Dance is a high-level choreography description language for modeling interaction models between participants [150]. It supports high-level modeling of choreographies and it is based on control flow and service interaction patterns. Let's Dance supports two different diagram types for modeling choreographies, including global models and local models. Global models are interactions defined from a viewpoint of an observer who see all interactions among services, while local models show only those interactions of a particular service.

Let's Dance model elementary interactions (message exchanges) between participants. Those interactions are building blocks for more complex interactions (choreographies) [146]. An elementary interaction represents a combination of a send activity model and a receive activity model, where an actor reference belongs to a role given for each activity model. Such a reference shows which activity instances must be performed by the same participant (usually one per role in a conversation).

In Figure 36, we show a description of elementary interactions in Let's Dance. This interaction is defined between a participant role 1 and a participant role 2, and it defines that a message type is sent during the interaction. This interaction defines a condition that evaluates if an elementary interaction is valid.

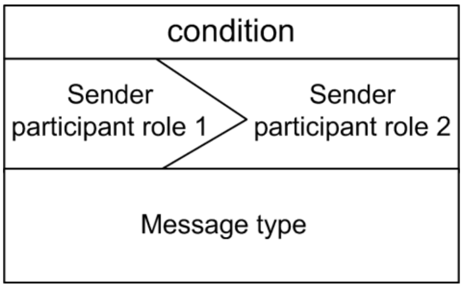


Figure 36. Elementary interactions in Let's Dance

There are four basic interaction constructs in Let's Dance, as shown in Figure 37. The first interaction shown in Figure 37a is *Precedes*, and it defines that after a receipt of a message (Message type) by participant role 2, participant role 2 is able to send a message (Message type 2) to sender participant role 1. Figure 37b shows the *Inhibits* relationship, with a crossed directed line, and the relationship defines that after sender participant role 2 receives the message (Message type) from participant role 1, sender participant role 2 cannot send the message (Message type 2) to participant role 1. In the case when two interactions inhibit each other, these situations are handled with the *Vice-versa-inhibits* relation shown in Figure 37c. Finally, relation called *WeakPrecedes* shown in Figure 37d denotes that participant role 2 cannot send a message (Message type 2) until participant role 1 has sent a message (Message type), i.e., when a source interaction has reached the "completed" status.

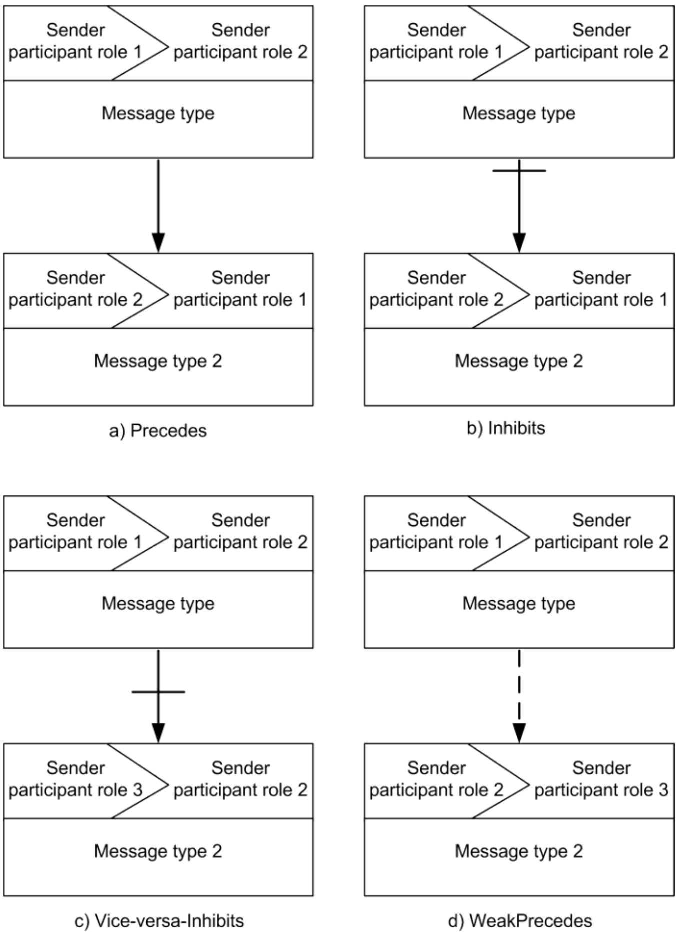

Sender participant role 1
Sender participant role 2
Message type
Sender participant role 2
Sender participant role 1
Message type 2
a) Precedes
Sender participant role 1
Sender participant role 2
Message type
Sender participant role 2
Sender participant role 1
Message type 2
b) Inhibits
Sender participant role 1
Sender participant role 2
Message type
Sender participant role 3
Sender participant role 2
Message type 2
c) Vice-versa-Inhibits
Sender participant role 1
Sender participant role 2
Message type
Sender participant role 2
Sender participant role 3
Message type 2
d) WeakPrecedes

Figure 37. Basic control flow constructs in Let's Dance

Along with basic control flow constructs, Let's Dance also support some advanced control flow constructs, where multiple interactions can be part of a composite interaction [146]. This language is not represented by a metamodel.

2.3.3.7. iBPMN

Decker et al. [22] extended the BPMN language with additional constructs to make it possible to model choreographies without shortcomings of the standard BPMN language, such as deadlocks or redundancies. They named the extended language - iBPMN. Redundancies are discovered in BPMN where modelers need more time to create and understand the models because branching, loops, and timeout events are duplicated in the model, as reported in [22]. Deadlocks are identified in situations when one participant waits for the other one to respond, while that participant also waits for the first participant to start some action; in such a case both the participants would wait endlessly.

iBPMN model interactions among parties, where each interaction is attached to a message flow in iBPMN, as shown in Figure 38. In iBPMN, pools are empty and only interactions among pools are shown. In this scenario, several bidders are involved in an auction, and in order to denote that it is possible to have multiple bidders, the shadowed pools (called participant sets) concept is introduced (the “Bidder” pool). At the start of the auction scenario, the Seller begins the auction. In this case, an explicit choice is used to model that where a participant decides which branch to take by using a data-driven XOR gateway. In addition, there is an association between the gateway and one of the pools in order to define who is responsible for carrying out the choice. Another important extension in iBPMN is the concept of participant reference passing. In Figure 38, the Seller needs to pass the Payment service reference to the Bidder, so that the Bidder can know what Payment service to use for the payment process. This reference passing is represented by using a data object attached to the message flow and with the corresponding participant.

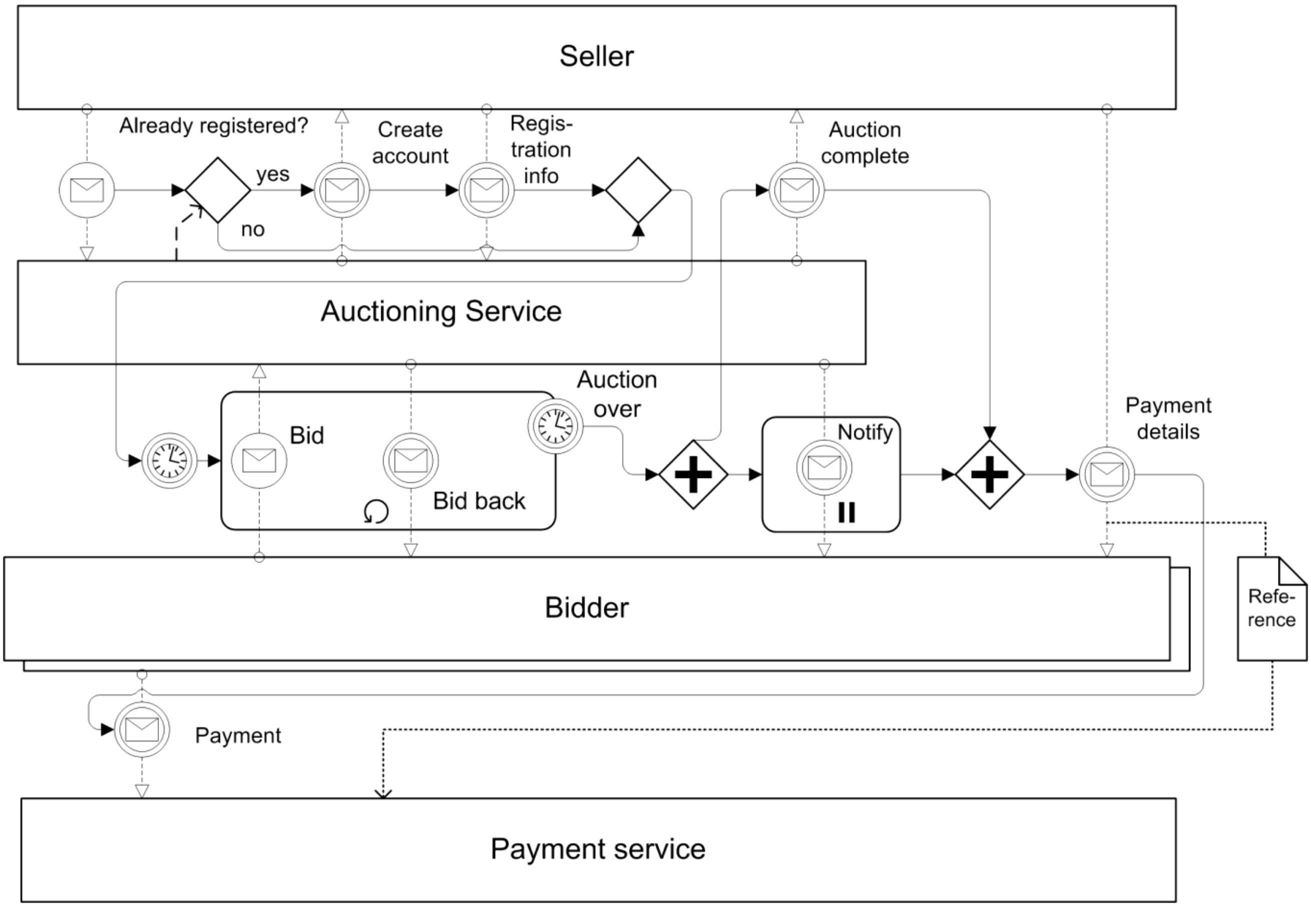


Figure 38. iBPMN interaction model for auction scenario (adapted from [22])

2.3.3.8. Business Process Modeling Notation (BPMN)

The BPMN is a de-facto standard for modeling business processes and it is created by OMG to be easily understandable for all business users [88]. BPMN has a graphical concrete syntax, but has no specific metamodel standardized in version 1.2, just a mapping to the Business Process Definition Metamodel [90]. Currently, there are also two possible options for BPMN language metamodel, and we analyze them in this chapter.

2.3.3.8.1. BPMN Language: Graphical Concrete Syntax

BPMN represents an OMG adopted specification [88] whose intent is to model business processes. The current version of BPMN is 1.2 [88], while a major revision process for BPMN 2.0 is in progress [89]. The later also includes a proposal for BPMN metamodel. In this thesis we will use BPMN 2.0 Beta 2 specification [89]. BPMN identifies the best practices of existing approaches and combines them into a new, generally accepted business process modeling language. Business process models are expressed in business process diagrams. Each business process diagram consists of a set of modeling elements. The details and different types for each group of BPMN modeling elements are given in Table I.

The BPMN in version 2.0 includes three types of flow objects that define behavior: Events, Activities and Gateways. In Figure 39 we show high-level structure of a BPMN as a mind map.

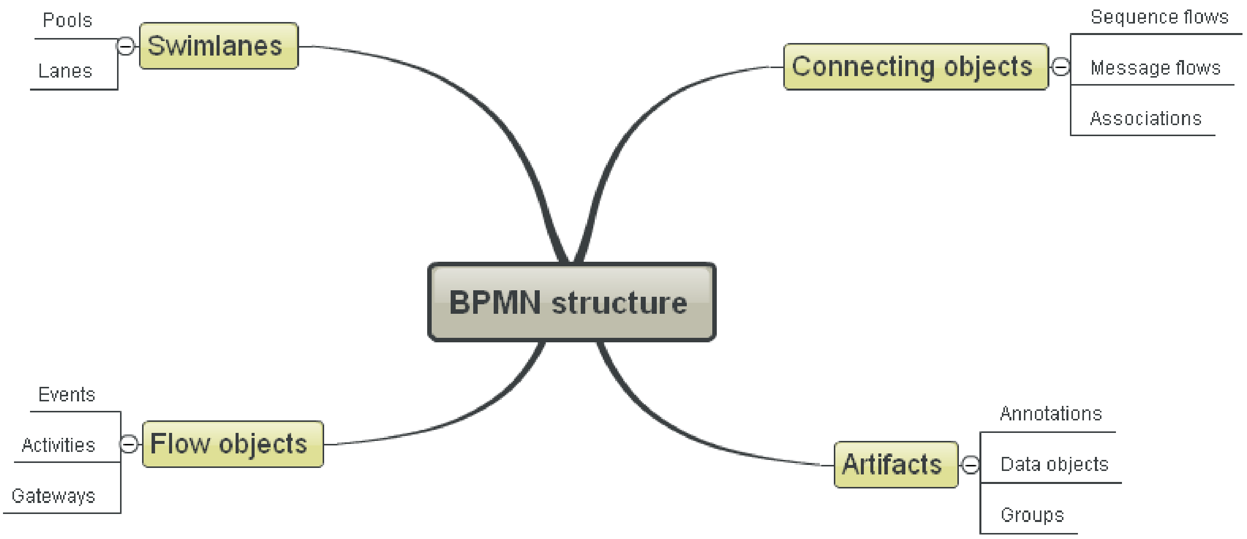


Figure 39. BPMN high-level structure

2.3.3.8.1.1. Events

An Event in BPMN is defined as “something that ‘happens’ during the course of a business processes” [88]. Events can be partitioned into three types, based on their position in the business process: start events are used to trigger processes, intermediate events can delay processes, or they can occur during processes [89] [146]. End events signal the termination of processes. The notational elements for the event trigger types are shown in Figure 40.

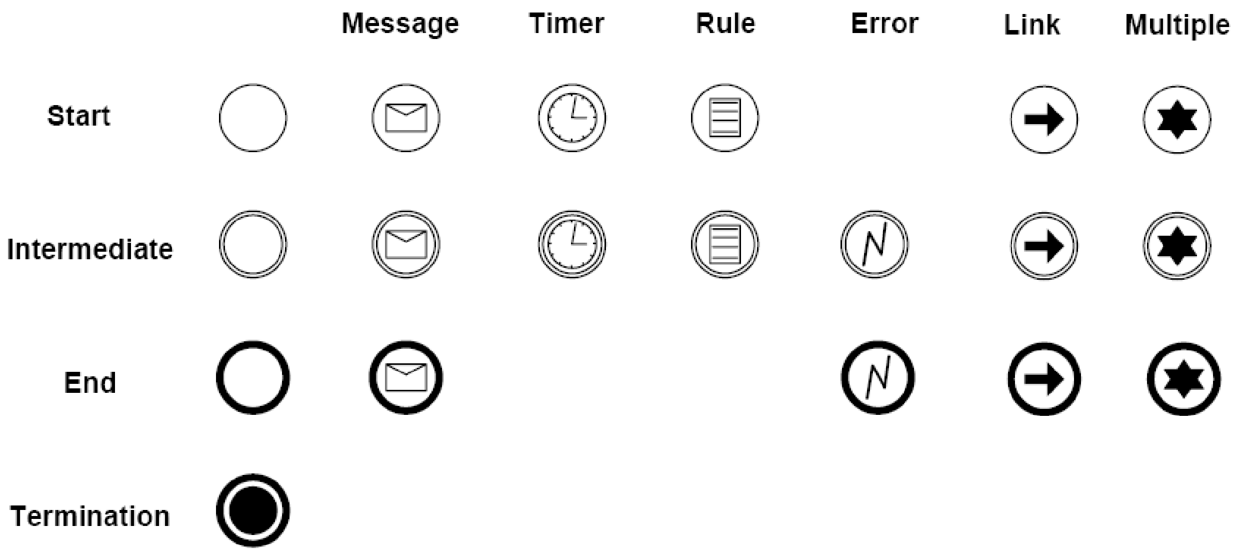


Figure 40. Event types in the BPMN [89]

Start events can have different triggers:

- *None*: No specific event trigger type is given. This is used when a subprocess is started by its parent process.
- *User*: A user manually starts a process, creating the start event of the process.
- *Message*: A message is received by a participant. The receipt of the message is then represented by a message event.
- *Timer*: A specific date or a specific cycle (e.g., every Monday at 9 a.m.) can be set that will trigger the start of the process.
- *Rule*: This event type is triggered when the rule evaluates to true.

Intermediate events occur during business processes. They are used to delay the execution of a process, for instance, to wait for a message to arrive. Intermediate events are also used to represent exception handling.

- *None*: Can be used to signal a state change in the process.
- *Message*: A step in the process is reached where progress depends on a message arriving from a participant. When the message arrives, the process can continue.
- *Timer*: An intermediate event is triggered based on timer information: A relative time specification (“after 7 days”) or an absolute time specification (“next Monday at 9 p.m.”) is useful here.
- *Error*: An intermediate error event generates an exception during the normal flow of the process. The exception is named with a unique identifier.

Some intermediate events can be attached to the boundary of activities. This event is used to represent catching an exception. When an exception is caught, a respective activity is started.

The meaning of end events is rather obviou. The *none* marker signals the completion of the process or subprocess without additional information. An error end event can be used to raise an exception. It can be caught by an intermediate event in the same event context. A termination end event is used to immediately terminate all activities of a given process.

In addition to these event types, there are additional event types that apply to start events, intermediate events, and end events.

- *Link*: A link is a mechanism for connecting the end of one process to the start (start trigger event) of another.
- *Multiple*: This means that there are multiple ways of triggering the process, one of which suffices to start it. The attributes of the start event define which triggers apply.

2.3.3.8.1.2. Activities

An activity is “a work that is performed within a business process” [88]. It can be atomic or non-atomic. BPMN support three types of activities: Process, Sub-Process and Task, where the latest two have a graphical representation. Activities are represented by rectangles (with rounded corners). An example of a Task is shown in Figure 41. Task has a TaskType attribute with default value None, but it can be one of the following types: Send, Receive, User, Script, Manual, Reference, and Service.

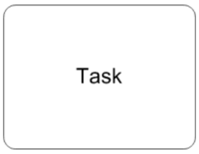


Figure 41. An example of a Task

2.3.3.8.1.3. Gateways

Gateways are defined as “modeling elements that are used to control how Sequence Flow interact as they converge and diverge within a Process” [88]. They are used for guiding, splitting and merging control flow. The diamond shaped gateways represent decisions, merges, forks, and joins in the control flow. A gateway can be thought of as a question that is asked at a point in the process. The question has a defined set of alternative answers, which are in effect gates. BPMN have two types of Exclusive gateways, the Event-based XOR gateway, which represents a branching point where the alternatives are based on an event that occurs at that point in the process flow, and the Data-based gateway, where alternatives are chosen based on defined condition. BPMN also has the Inclusive gateway where more than one possible alternative is possible, the Parallel gateway where multiple parallel paths are possible and the Complex gateways are branches in a Process where more advanced behavior can be defined.

2.3.3.8.1.4. Connecting objects

Connecting objects (i.e., different kinds of lines) connect the flow objects to create a basic structure of a business process. A Sequence Flow is represented by a solid arrow and is used to show the order that activities will be performed in a business process. A Message Flow is represented by a dashed line with an open arrowhead and is used to show the flow of messages between two separate business process participants. Associations, represented as dotted lines, are used to associate data objects, text, and other artifacts with flow objects. Message flows and sequences flow can be connected with other BPMN objects only by following Message Flow and Sequence Flow rules [88].

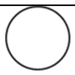
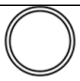
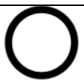
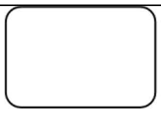
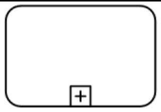
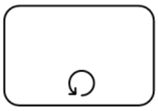
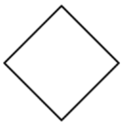
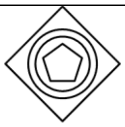
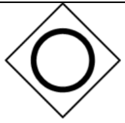
2.3.3.8.1.5. Swimlanes

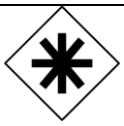
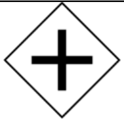
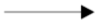
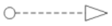
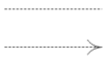
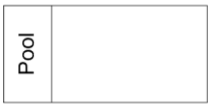
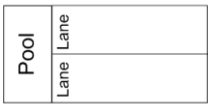
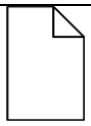
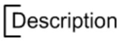
BPMN also has a concept called a Pool, which represents a participant in the process. A participant can be a specific business entity (e.g., a company) or can be a more general business role (e.g., buyer or seller). Graphically, a Pool is a container for partitioning a process from other Pools, when modeling business-to-business situations, although a Pool might not need to have any internal details (i.e., it can be a “black box”) [88]. Every Pool can have multiple Lanes and they partition Pool’s in order to organize activities within a Pool. They are often used to represent internal roles.

2.3.3.8.1.6. Artifacts

Artifacts are additional information added into a Process. BPMN have three types of artifacts: a Data Object, a Group and an Annotation. A data object shows how documents and data are used in a Process. Text annotations are used in BPMN to define additional information on a BPMN diagram, and they are connected with an association with a specific object. Groups are used to group BPMN elements informally.

Table I. BPMN main elements

Group		Element	Description
Flow Objects	**Events**	 Start	Start Event indicates where a Process will begin.
		 Intermediate	Intermediate Event occurs after a process has started and before a process ended.
		 End	End Event indicates where a process will end.
	Activities	 Task	A Task is an atomic activity that is included within a Process. A Task is used when the work in the Process is not broken down into a finer level of Process Model detail.
		 Sub-Process	Sub-Processes enable for hierarchical Process development. A Sub-Process is a compound activity that is included within a Process. It is compound in that it can be broken down into a finer level of detail (a Process) through a set of sub-activities.
		 Looped Task	Looped Task is an activity that is repeated (looped). There are two types of loops: Standard and Multi-Instance, where Multi-Instance are activities that are copied a needed number of times. Those activities that are Parallel Multi-Instance have a parallel marker placed in the bottom center of the activity shape ‖.
	Gateways	 Exclusive (Data-based)	An exclusive gateway is location within a business process where a Sequence Flow can take two or more alternative paths. This is basically a “fork in the road” for a process. Only one of the possible outgoing paths can be taken when the Process is performed. Data-based exclusive gateways create alternative paths based on defined conditions (i.e., condition expressions), and they can be shown with or without an internal “X” marker.
		 Exclusive (Event-based)	This type of Decision represents a branching point in the process where the alternatives are based on events that occur at that point in the Process, rather than conditions. The Event that follows the Gateway Diamond determines the chosen path, where the first Event triggered wins.
		 Inclusive	Inclusive Gateways are Decisions where there is more than one possible outcome.

		 Complex	Complex Gateways are Decisions where there are more advanced definitions of behavior that can be defined.
		 Parallel	Parallel Gateways are places in a Process where multiple parallel paths are defined.
Connectivity objects		 Sequence Flow	A Sequence Flow is used to show the order that activities will be performed in a Process.
		 Message Flow	A Message Flow is used to show the flow of messages between two entities that are prepared to send and receive them.
		 Association	An Association is used to associate data, information and artifacts with flow objects.
Swimlanes		 Pool	Pools represent Participants in an interactive (B2B) Business Process Diagram.
		 Lane	Lanes represent sub-partitions for the objects within a Pool.
Artifacts		 Data object	Data Objects are Artifacts that are used to show how data and documents are used within a Process. Data Objects can be used to define inputs and outputs of activities.
		 Group	Groups are Artifacts that are used to highlight certain sections of a Diagram without adding additional constraints for performance – as a Sub-Process would.
		 Text annotation	Text Annotations are a mechanism for a modeler to provide additional information about a Process. They can be connected to a specific object on the Diagram with an Association.

2.3.3.8.2. BPMN Metamodel: Abstract Syntax

As our work is based on MDE principles, and as BPMN language does not have a metamodel in current specification v1.2 [88], in this section, we first define a set of criteria for choosing an adequate BPMN metamodel.

2.3.3.8.2.1. Choosing BPMN metamodel for BPMN language

In our research, we need to start from one of the existing well-known proposals for the BPMN metamodel [50] [89] [90]. Choosing a metamodel that will satisfy our criteria will start from defining those criteria. For choosing an appropriate business process metamodel, we decided to use the criteria defined in [78], which are defined from the analysis of 15 specifications the authors of [78] gathered. The following list of 13 high-level metamodel concepts reflects criteria from [78]:

1) *Task I/O*: The term task is used to refer to basic units of work whose temporal and logical relationships are modeled in a process. The input and output (I/O) of these tasks may be modeled using simple or XML complex types.
2) *Task Address*: The address specifies where or how a service can be located to perform a task. The address can be modeled directly via a URI reference of a service or indirectly via a query that identifies a service address.
3) *Quality Attributes*: When a set of potential services is generated via a query, quality attributes may be used to identify the “best” service.
4) *Task Protocol*: The protocol defines a set of conventions to control interaction with a service performing a task. Web Services use SOAP as a protocol.
5) *Control Flow*: The control flow defines the temporal and logical relationships between different tasks. Control flow can be specified via directed graphs, block oriented nesting of control instructions, or process algebra.
6) *Data Handling*: Data handling specifies which variables are used in a process instance and how the actual values of these variables are calculated.
7) *Instance Identity*: This concept addresses how a process instance and related messages are identified. Correlation uses a set of message elements that are unique for a process instance in order to route messages to process instances. The generation of a unique identifier which is included in the message exchange is an alternative approach.
8) *Roles*: Roles provide for an abstraction of participants in a process. Roles are assigned to tasks and users to roles. A staff resolution mechanism can then allocate tasks of a process instance to users.
9) *Events*: Events represent real-world changes. Respective event handlers provide the means to respond to them in a predefined way.
10) *Exceptions*: Exceptions or faults describe errors during the execution of a process. In case of exceptions, dedicated exception handlers undo unsuccessful tasks or terminate the process instance.
11) *Transactions*: ACID transactions define a short-run set of operations that have all-or-nothing semantics. They have to be rolled back when one partial operation fails. Business transactions represent long-running transactions. In case of a failure, the effects of a business transaction are erased by a compensation process.
12) *Graphic Position*: The graphical presentation of a business process model contributes to its comprehensibility. The attachment of graphical position information can be an explicit part of the metamodel.
13) *Statistical Data*: Performance analysis of a business process builds on statistical data such as costs or duration of tasks.

We extend these criteria with some of the concepts defined in [126] for the area of extension. This extension will include rules, policies and vocabularies, and a support for choreography and orchestration. The extended metamodeling concepts (adapted from [126]) are:

1) *Understandability*: denotes the users' effort recognize modeling concepts, language, notation and the entire metamodel. Conceptual understandability is a subjective assessment of the modeling

method’s elements and constructs, which may sometimes be in conflict with the model’s syntactical correctness.

2) *Clarity:* refers to representational issues such as a structure, a graphical representation, and a general readability of a metamodel. Clarity means unambiguity and distinction between concepts. In a visual representation, clarity means using the minimal symbolism necessary to represent some structural or behavioral property on the diagram (i.e. relevance).
3) *Extensibility:* means that a metamodel is extendable with new concepts or some existing concepts are further developed. Extensions should not cause revisions of existing definitions, i.e. the metamodel should be consistent after extensions.
4) *Adaptability*: is the opportunity for specialization and adaptation of modeling concepts and methods to different domains and organizations.
5) *Operability:* is the ability to implement the modeling framework or the metamodel and to use them operationally.

Table II gives an overview of our analysis. A plus sign indicates that a concept mentioned in the first column is included in the metamodel of the proposal mentioned in the headings of the columns. A minus sign denotes that the concept is not included.

Table II. Overview of BPMN metamodel proposals

	BPDM [90]	**BPMN2** [89]	**Intalio BPMN** [50]
Task I/O	+	+	+
Task Address	+	+	+
Quality Attributes	-	-	-
Protocol	+	+	+
Control Flow	+	+	+
Data Handling	+	+	+
Instance Identity	+	+	-
Roles	+	+	+
Events	+	+	+
Exceptions	+	+	+
Transactions	+	+	-
Graphic Position	-	-	-
Statistical Data	+	-	-
Understandability	-	+	+
Clarity	-	+	+
Extensibility	+	+	-
Adaptability	+	-	-
Operability	-	+	-

We now discuss each proposal in more detail.

1) **BPDM**: OMG’s Business Process Definition Metamodel (BPMD) [90] is a proposal for the metamodel of the BPMN2 language. BPDM provides BPMN with a metamodel, a serialization mechanism (XML) and execution semantics. BPMD is composed of elements that use the UML 2 Infrastructure elements [96]. BPMD can be used to represent other types of business processes languages, not only BPMN. The lack of this proposal is a high-complexity and peculiar terminology. Through different packages and a lot of abstract concepts, it is hard to follow BPMN concepts, while the terminology used is not clearly mappable to the BPMN concepts. For example,

the BPMN Pool is called “processor role”, while a sequence flow is called “Succession”. There are also some other unclear concepts, like the difference between “Fact condition” and “Fact change condition”. The authors of [90] also claim that Statements and Fact Conditions in the Condition package are used to integrate with rule models, but it is not clear in which way this integration can be done.

2) **Intalio BPMN**: this is not actually an official proposal for a BPMN metamodel, but it is a BPMN metamodel used in their BPMS Designer [50]. This metamodel represents a simple subset of BPMN concepts used in Intalio’s tool, and as we can see from Table II it lacks from different metamodeling concepts from [78]. This metamodel also does not have any documentation, which hampers its extensions and further adaptation for different needs.
3) **BPMN2**: OMG’s Business Process Model and Notation (BPMN) v2.0 Beta 2 [89] is the latest proposal for the BPMN metamodel in the BPMN language lead by BEA Systems, IBM, Oracle and SAP. This proposal uses an explicit BPMN terminology; it is much simpler than BPDM [90] (much less abstract classes are used); and it is clearly mappable to BPEL. This proposal also has an XML-based serialization for BPMN models (XMI). For example, in this metamodel proposal, the BPMN sequence flow element is actually represented with the BPMN concept “SequenceFlow”. For extensions, the authors of the proposal defined the BPMN Extensibility Model that allows BPMN adopters to extend the specified metamodel in a way that allows them to be still BPMN-compliant.

From the Table II, we can see that the third proposal (BPMN) supports the most of the metamodeling concepts from [78], and for the reasons described above, such as mappability to BPEL, easy extensions and XML format, we have choose this metamodel BPMN language.

2.3.3.8.2.2. BPMN metamodel

In this subsection, we describe in detail the selected metamodel to be used later in our definition of the Rule-enhanced Business Process Modeling Language. The BPMN specification [89] is structured in layers, where each layer builds on top of and extends lower layers. Here, we discuss the Core package of the BPMN metamodel, which includes the most fundamental elements that are required for modeling the flow of activities, events, messages, and how they are sequenced. The Core package contains four sub-packages (see Figure 42):

- *Common*: Those classes are common to multiple packages.
- *Foundation*: The fundamental constructs needed for modeling of Processes.
- *Service*: The fundamental constructs needed for modeling services and interfaces.

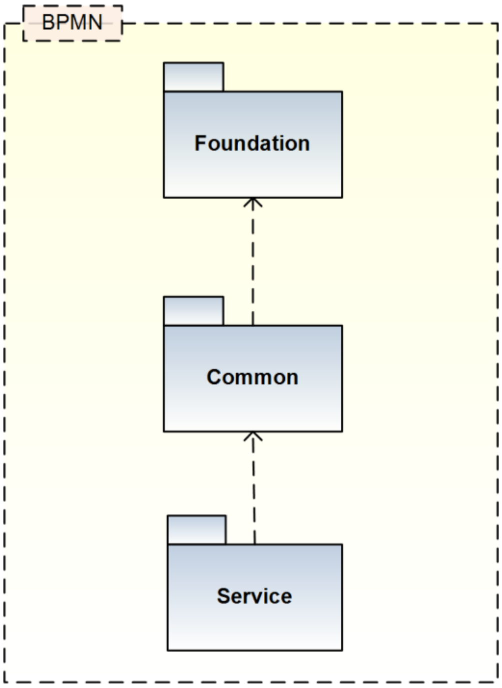


Figure 42. Core package

The Core package in the BPMN metamodel [89] is shown in Figure 43. This core package consists of the following elements:

- *Process* (concept - class) describes a sequence or flow of activities in an enterprise with the objective of carrying work. In BPMN, a *Process* is depicted as a graph of Flow Elements, which is a set of activities, events, gateways and sequence flows that define finite execution semantics (see Figure 45).
- *Collaboration* is used to describe interactions between two or more business entities or business roles, which are represented as *Participants* within *Pools*. A *Collaboration* shows interactions, that is, *Messages* exchanged among *Participants* that take part in the *Collaboration*. A collaboration contains two or more *Pools*, representing the *Participants* in the *Collaboration*. The interactions among *Participants* are shown by *MessageFlow* that connect two *Pools*.
- *MessageFlow* connect either to the *Pool* boundary or *Flow* objects within the *Pool* boundary (they are represented as dashed lines with an arrow on the one side, and a circle on the other side). Every *MessageFlow* can have zero or one *Message* attached (see Figure 43).
- *Pool* represents a *Participant* in a *Collaboration*. A *Participant* can be a specific business entity. Every *Participant* has a partner *RoleRef* attribute (of enumeration type *Role*) that we have added and that defines a business role that the *Participant* plays in the *Collaboration* (such as web service – << WS >> and BPEL process – << BpelProcess >>). A *Pool* acts as the container for a *Process*.

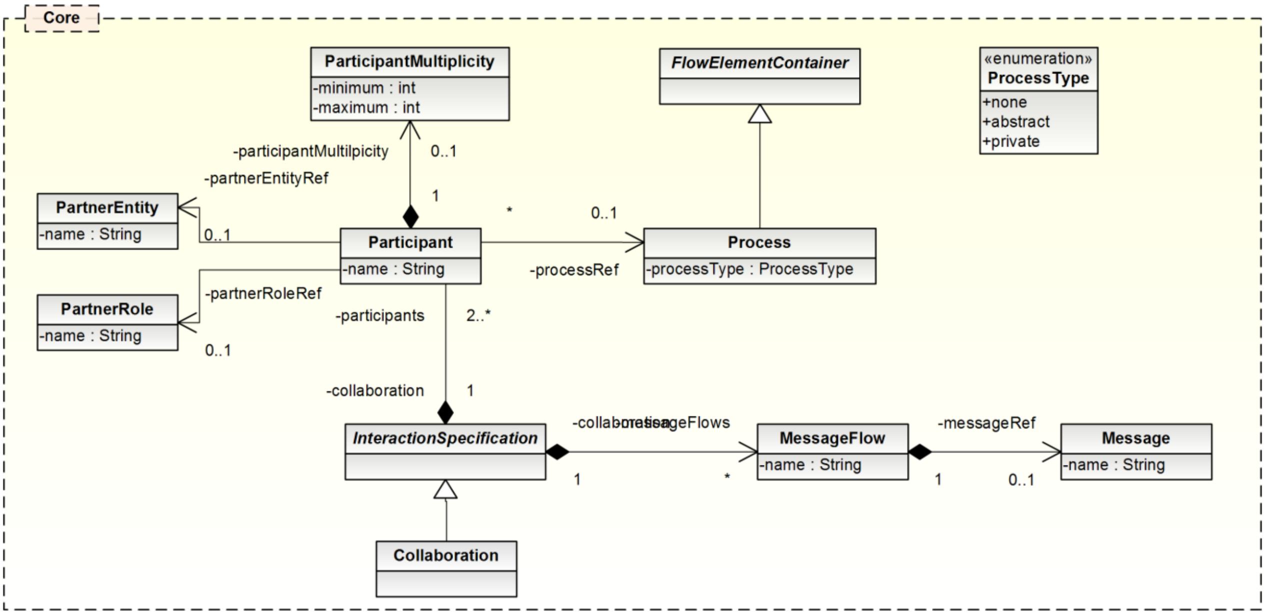


Figure 43. Core package in the BPMN metamodel (an excerpt)

The Common package contains classes that are shared amongst other packages in the Core (see Figure 44).

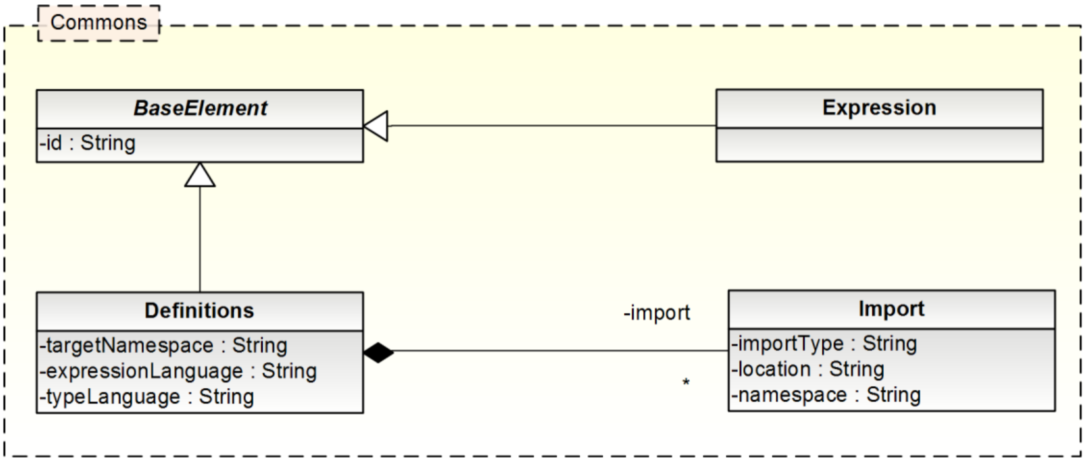


Figure 44. Classes in the Common package (an excerpt)

BaseElement is the abstract superclass for all BPMN elements, where *ReusableElement* is the abstract superclass for all BPMN elements that are can be referenced across *Definitions*. Examples of concrete reusable elements include *Process*, *Collaboration*, and *Message*. The *Definitions* class is the outermost containing object for all BPMN elements. It defines the scope of visibility and the namespace for all contained elements. The interchange of BPMN files is done through one or more *Definitions*. The *Import* class is used when referencing external element, either BPMN elements contained in other BPMN *Definitions* or non-BPMN elements.

The *Process* package contains classes which are used for modeling the flow of activities, events, messages, and how they are sequenced within a *Process* (see Figure 45).

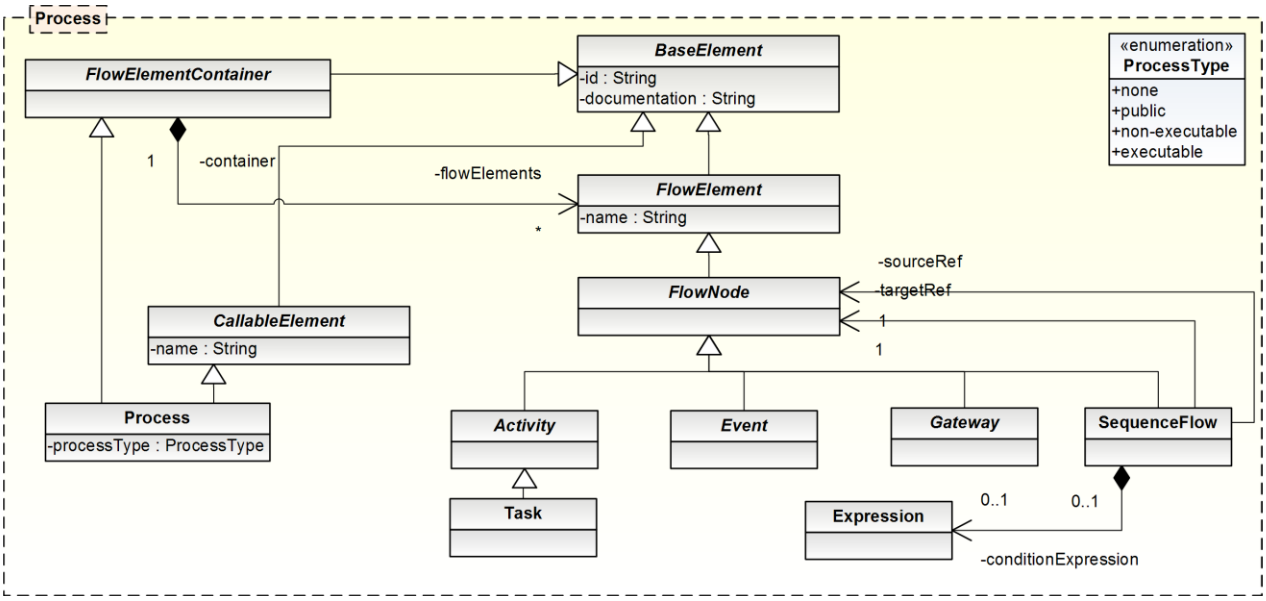


Figure 45. Process package in BPMN metamodel (an excerpt)

This package consists from different elements (classes) from:

- A *SequenceFlow* is used to model the transition of control from one *FlowElement* (the source) to another (the target). It determines the sequencing of *FlowNodes* within a *Process* flow. A SequenceFlow is represented by a solid line with a black arrow between *Tasks*. A *Sequence Flow* can optionally define a condition expression, indicating that the 'transfer of control' will only be available if the expression evaluates to true. This expression is typically used when the source of the *SequenceFlow* is a *Gateway* or an *Activity*.
- *Activities* represent points in a *Process* flow where work is performed. They are executable elements of a BPMN *Process*. The *Activity* class is an abstract element. The types of activities that are part of a *Process* are *Task*, *SubProcess*, and *CallActivity*. In Figure 45, we show only the *Task* element, as the most important activity type. A *Task* is an atomic *Activity* within a *Process* flow, which is used when the work in the *Process* cannot be broken down in to a finer level of detail. Generally, an end-user and/or applications are used to perform the *Task* when it is executed.
- The Process package also includes *Events* and *Gateways*. *Gateways* are used to control how *SequenceFlows* interact as they converge and diverge within a *Process*. If the flow does not need to be controlled, then a *Gateway* is not needed. The term "Gateway" implies that there is a gating mechanism that either allows or disallows passage through the *Gateway*.
- An *Event* is something that "happens" during the course of a Process. These *Events* affect the flow of the *Process* and usually have a cause or an impact and in general require or allow for a reaction. In BPMN, there are different types of start, intermediate and end events [89].

Activities represent points in a *Process* flow where work is performed. They are the executable elements of a BPMN *Process*. The *Activity* class is an abstract element, subclassing from the *FlowNode* (as shown in Figure 46). The *Activity* class is the abstract super class for all concrete activity types. The *Performer* class defines the resource that will perform or will be responsible for an activity. The performer can be specified in the form of a specific individual, a group, an organization role or position, or an organization. The *CallableElement* is the abstract super class of all activities that have been defined outside of a *Process* but which can be called (or reused) from within a *Process* flow. It references an *Interface* that specifies the external behavior of the element being called. *Callable Elements* are reusable elements, which can be imported and used in other *Definitions*.

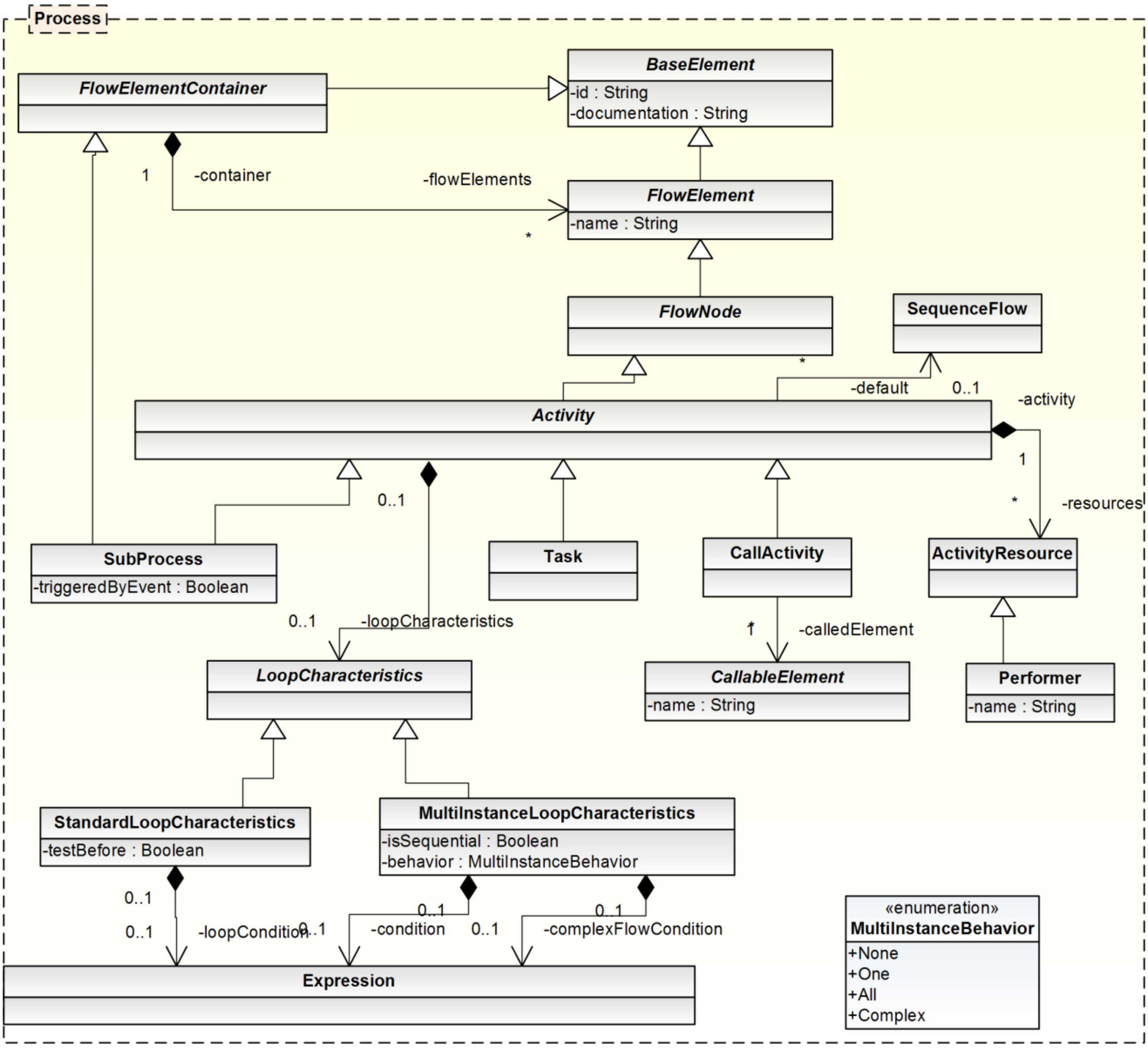


Figure 46. Activity classes in BPMN (an excerpt)

Gateways are used to control how *Sequence Flows* interact as they converge and diverge within a *Process*. If a flow does not need to be controlled, then a Gateway is not needed. The term “Gateway” implies that there is a gating mechanism that either allows or disallows passage through the *Gateway*-that is, as Tokens arrive at a *Gateway*, they can be merged together on input and/or split apart on output as the *Gateway* mechanisms are invoked. A *Gateway* controls the flow of both diverging and converging *SequenceFlow*. That is, a single *Gateway* could have multiple input and multiple output flows. Thus, it would take two sequential *Gateways* to first converge and then to diverge the *SequenceFlow*. The Gateway class diagram is shown in Figure 47 and details for the types of Gateways (Exclusive, Inclusive, Parallel, Event-Based, and Complex) is defined in Section 2.3.3.8.1.3.

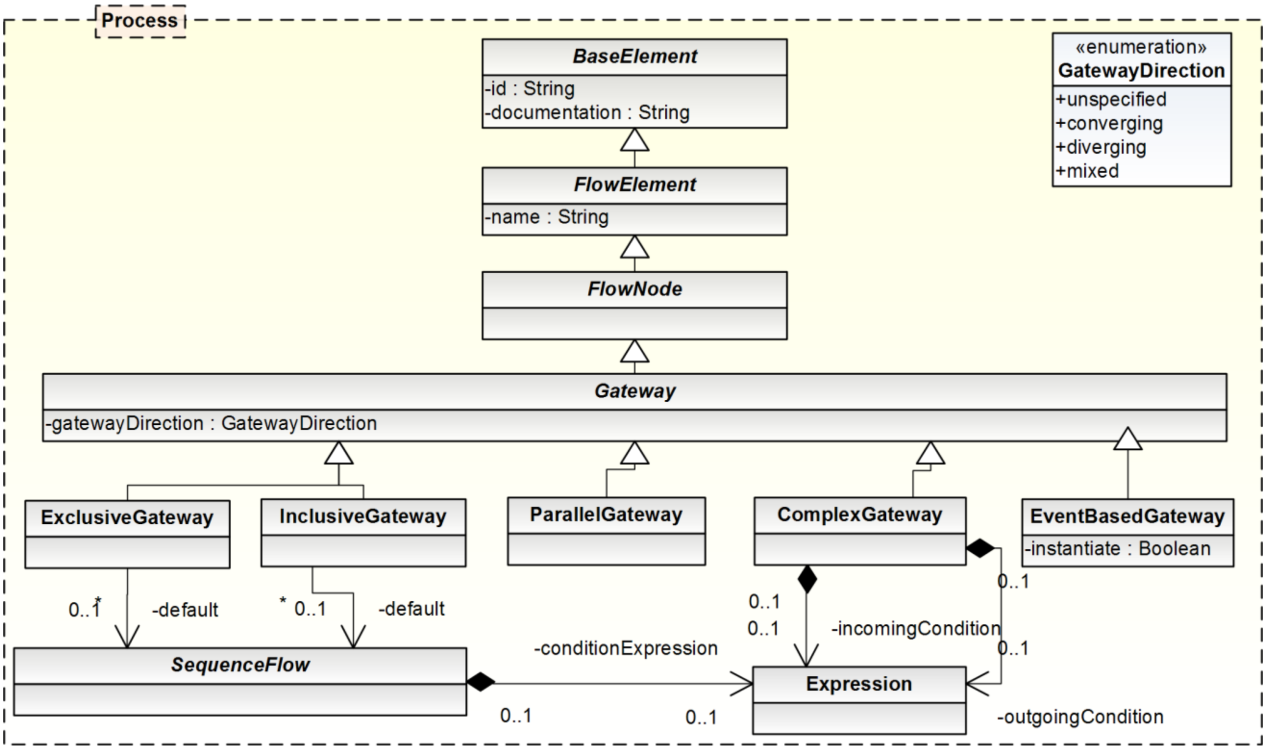


Figure 47. Gateways in BPMN metamodel (an excerpt)

The *Collaboration* package contains classes that are used for modeling *Collaborations*, which is a collection of *Pools*, their interactions as shown by *MessageFlow*, and may include *Processes* and/or *Choreographies* (see Figure 48). A *Participant* is a specific business entity (e.g., a company) or a more general business role (e.g., a buyer, seller, or manufacturer) responsible for the execution of the *Process* enclosed in a *Pool*. *Participants* may also be defined for pools that do not contain a *Process*. *ParticipantMultiplicity* is used to define the multiplicity of a *Participant*.

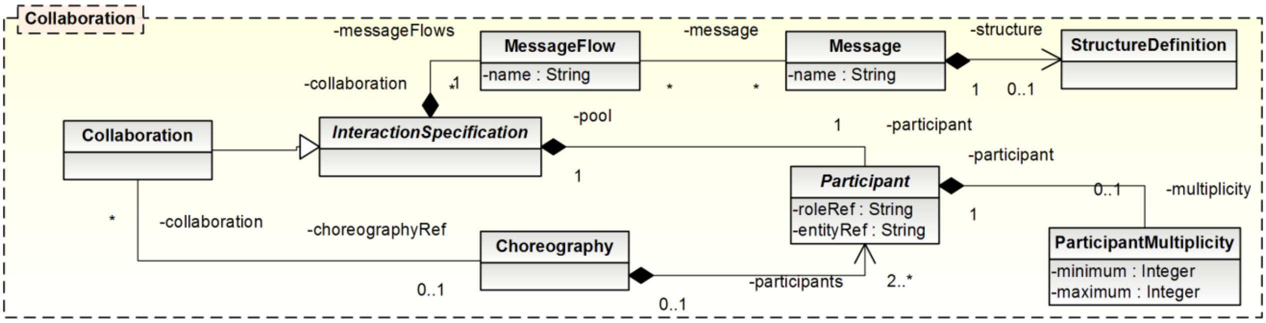


Figure 48. Collaborations package in BPMN metamodel (an excerpt)

A *Choreography* defines business a contract between two or more interacting participants. While an Orchestration Process exists within a BPMN *Pool*, a *Choreography* Process exists between *Pools*. A *Choreography* Process defines the order in which *Choreography* Tasks are executed. A *Choreography* Task is an atomic activity and represents a coherent set of *Message* exchanges. *Choreography* Sub-Processes allow the composition of *Choreographies*.

The BPMN metamodel is aimed to be extensible. This allows BPMN adopters to extend the specified metamodel in a way that allows them to be still BPMN-compliant. It provides a set of Extension elements (see Figure 49), which allows BPMN adopters to attach additional attributes and elements to standard and existing BPMN elements.

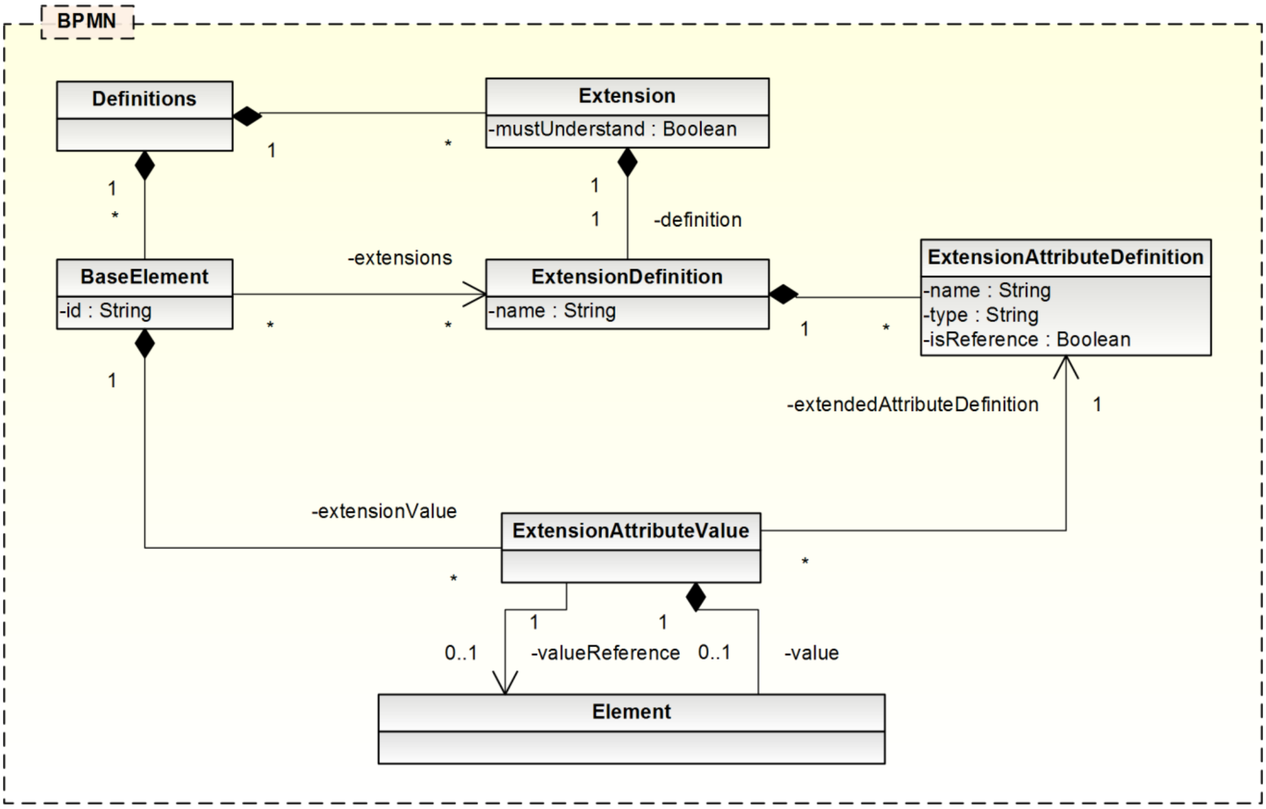


Figure 49. Extensions package in BPMN metamodel (an excerpt)

A BPMN Extension basically consists of four different elements:

- *Extension*
- *ExtensionDefinition*
- *ExtensionAttributeDefinition*
- *ExtensionAttributeValue*

The core elements of an *Extension* are the *ExtensionDefinition* and *ExtensionAttributeDefinition*. The latter defines a list of attributes that can be attached to any BPMN element. The attribute list defines the name and type of the new attribute. This allows BPMN adopters to integrate any metamodel into the BPMN metamodel and reuse already existing model elements. The *ExtensionDefinition* itself can be created independent of any BPMN element or any BPMN definition.

In order to use an *ExtensionDefinition* within a BPMN model definition (*Definitions* element), the *ExtensionDefinition* must be associated with an *Extension* element which binds the *ExtensionDefinition* to a specific BPMN model definition. The *Extension* element itself is contained within the BPMN element *Definitions,* and therefore it is available to be associated with any BPMN element making use of the *ExtensionDefinition*.

Every BPMN element which subclasses the BPMN *BaseElement* can be extended by additional attributes. This works by associating a BPMN element with an *ExtensionDefinition* which was defined at the BPMN model definitions level (element *Definitions*). Additionally, every “extended” BPMN element contains the actual extension attribute value. The attribute value, defined by the element *ExtensionAttributeValue* contains the value of type *Object*. It also has an association to the corresponding attribute definition.

Expressions are used in many places within BPMN to extract information from the model (see Figure 50). The most common usage is when modeling decisions, where conditional expressions are used to direct the flow along specific paths based on some criteria. BPMN supports underspecified expressions, where the logic is captured as natural-language descriptive text. It also supports formal expressions, where the logic is captured in an executable form using a specified expression language.

The *Expression* class is used to specify an expression using natural-language text. These expressions are not executable, while the *FormalExpression* class is used to specify an executable expression using a specified expression language.

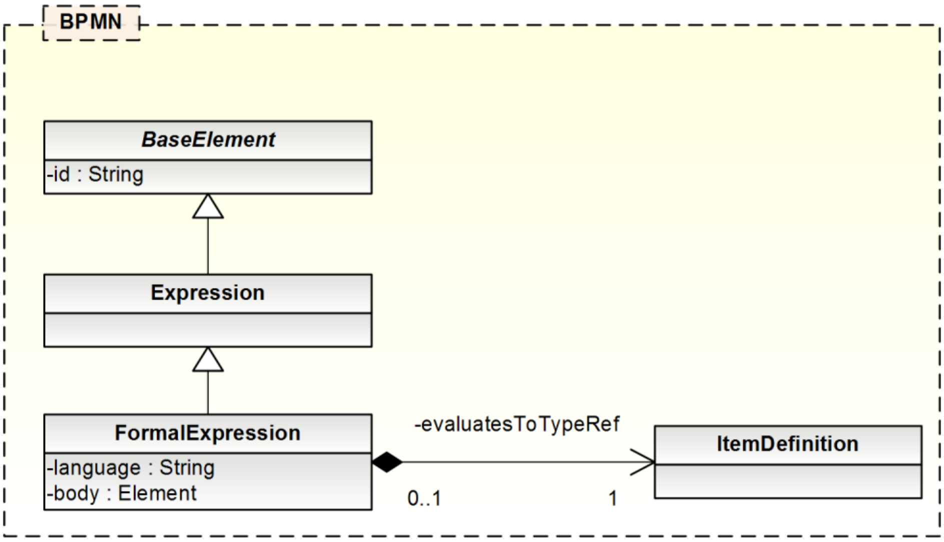


Figure 50. Expressions in BPMN metamodel (an excerpt)

An *Event* is something that "happens" during the course of a *Process*. The *Event* package is shown in and in Figure 51.

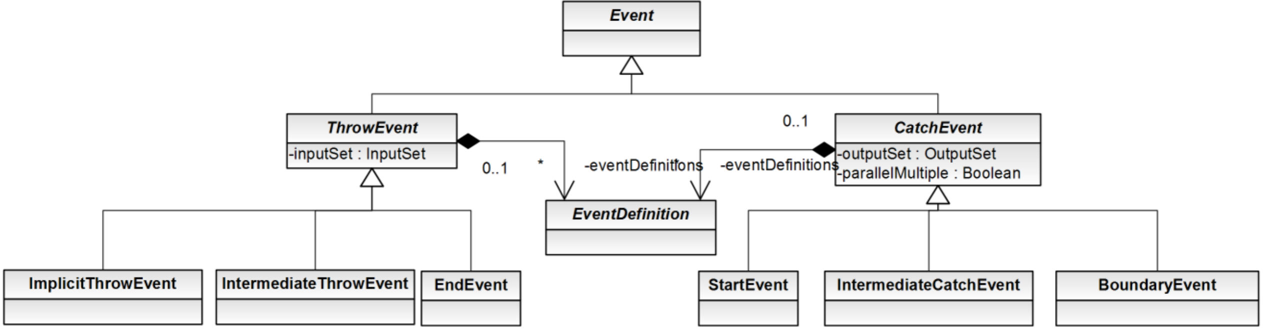


Figure 51. Events in BPMN metamodel (an excerpt)

The Event Definition BPMN metamodel package which represents types of events is shown in Figure 52. These Event Definition concepts are described in Section 2.3.3.8.1.1

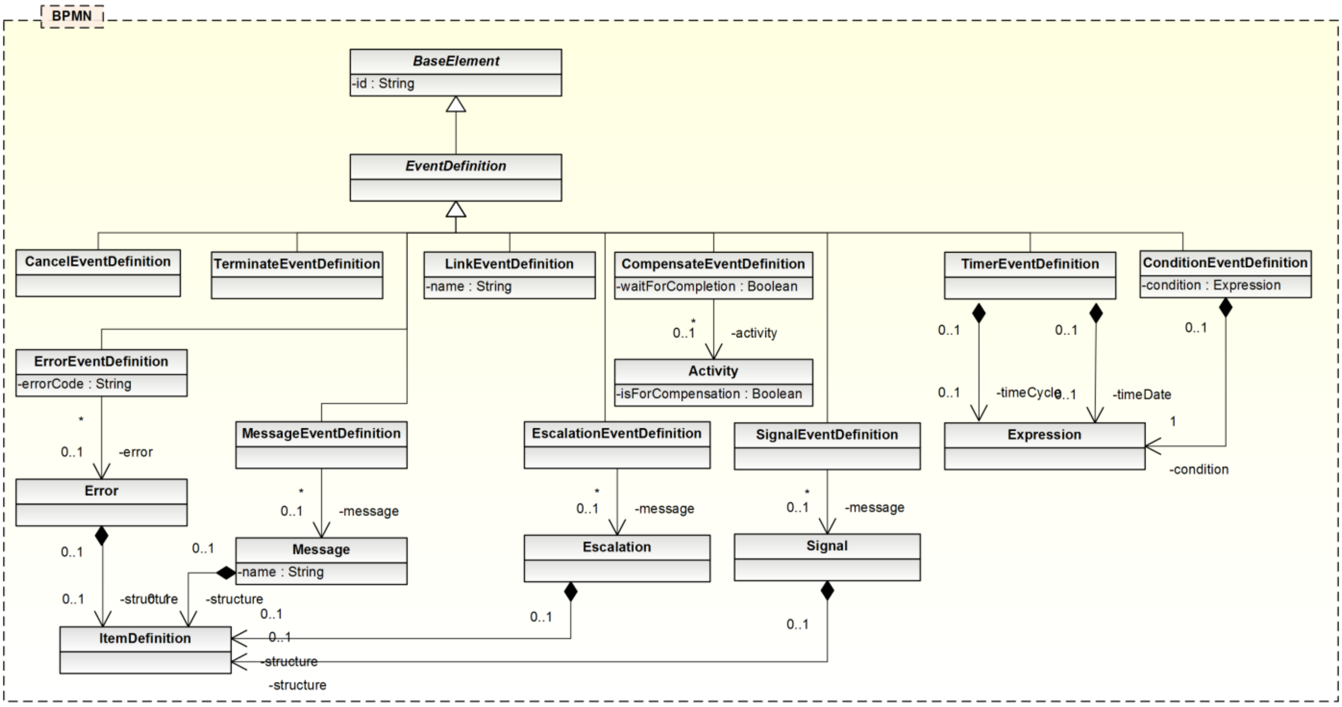


Figure 52. Event definitions in BPMN metamodel (an excerpt)

An important requirement in process modeling is ability to model the items that are created and manipulated during the execution of a process (see Figure 53). Basically, these items represent data objects in a process model, and in BPMN 2.0, data items (*ItemAwareElement*) are represented with following classes: Data Objects, ItemDefinition, Properties, Data Inputs, Data Outputs, Messages, Input Sets, Output Sets, and Data Associations.

Item-aware elements are used in a BPMN 2.0 process to (optionally) store items during process execution, just like a variable in programming languages. A structure of an Item-aware element is defined by using an *ItemDefinition*.

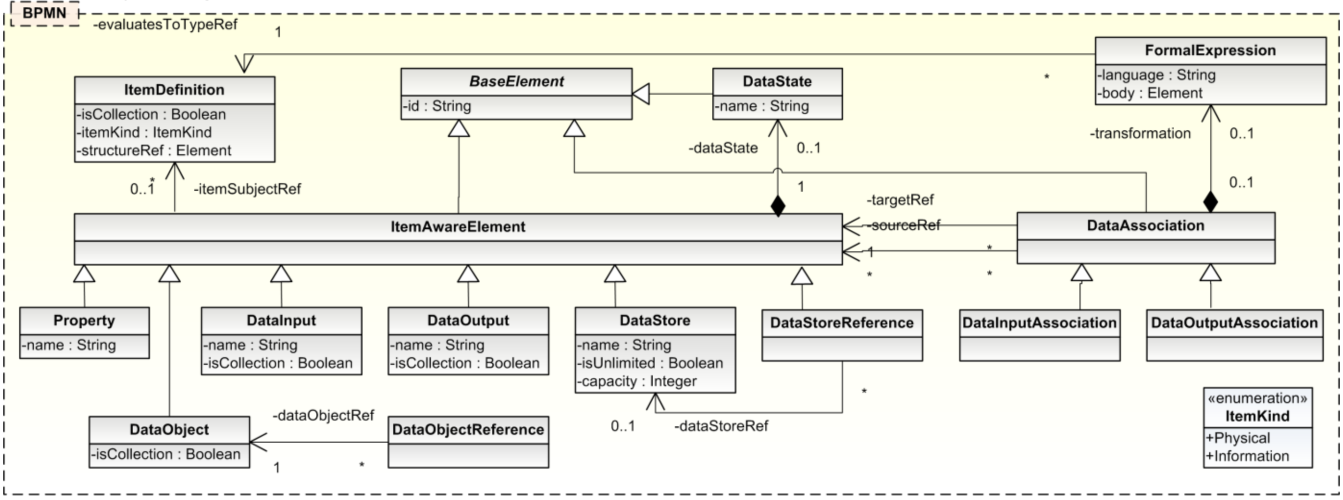


Figure 53. Item-aware elements in BPMN metamodel

Data objects are item-aware elements that are contained in a process, and represented graphically in a process diagram (see Figure 54), while *DataObjectReference* is used to reference to a data object in multiple points in a diagram.

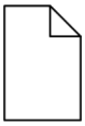


Figure 54. A DataObject in BPMN metamodel

DataStore is used to enable retreiving and storing infromations for activities, and they are persistant beyond the scoope of a process. *DataStore* is represented graphically in a process diagram as shown in Figure 55. *DataStoreReference* is used to reuse the same *DataStore* in a process diagram.

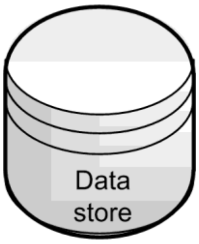


Figure 55.A DataStore in BPMN metamodel

DataInput's are used in a process to provide or produce data for an activity or a process. *DataInput* element is an item-aware element represented graphically on a process diagram to show the inputs of an activity (see Figure 56a). *DataOutput*'s are similar to *DataInput*'s, but they are used to show outputs of an activity. They are shown graphicall in Figure 56b.

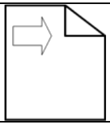
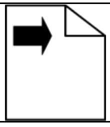
	
a) DataInput	b) DataOutput

Figure 56. DataInput and DataOutput elements in BPMN metamodel

Data associations are used to move data between data objects, and by doing this to fill activities input or to push the output values from an executed activity to an output data object. Data associations are used to push or pull data from item-aware elements, and they have one or more sources and a target. Data associations are represented by a dashed line with an arrow (see Figure 57).

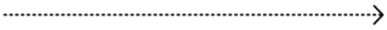


Figure 57. A DataAssociation in BPMN metamodel

The *DataInputAssociation* is used to association an Item-aware element with a *DataInput* from an activity, while *DataOutputAssociation* is used to associate appropriate *DataOutput* form an activity with an Item-aware element. In addition, *DataAssociation*'s can be associated to a Sequence flow, and this replaces two data associations, one from an Item-aware element to an data object, and another from an data object to an Item-aware element.

More details about data representation in BPMN 2.0, can be found in BPMN 2.0 OMG specification [241].

2.3.3.9. Business Process Modeling Languages Summary

In this section, we show an evaluation of business process modeling languages shown in Section 2.3.2. In Table III, we list an integrated and extended version of the evaluation of business process modeling languages from [63]. The first column describes if a language has a metamodel, the second column shows if the language has a graphical notation, the third column show if the language has a translation into executable code, the fourth column show if the language has a tool(s) to work with it, and the last column shows if the language support any types of rules.

Table III. Summary of BPML's (adapted from [63])

	Meta-model	**Notation**	**Execution language**	**Tool(s)**	**Rule support**
Petri Nets	+	+	PNML [11]	DaNAMiCS [21], Renew [65], Petri net Kernel (PNK) [101]	-
Event Driven Process Chain (EPC)	+	+	EPML [79]	ARIS Toolset [3], ADONIS [2]	+/-
Integrated DEFinition Method 3 (IDEF3)	-	+	-	Procap [106]	-
UML 2.0 Activity Diagrams (AD)	+	+	WS-BPEL [49]	MagicDraw UML [74], Microsoft Visio [83], MDT-UML2Tools [77]	-
Agent Object-Relationship Modeling Language (AORML)	+	+	JADE [8]	Integrated Business Process Editor [129]	Reaction rules
Let's Dance	-	+	-	The Oryx Editor [98]	-
iBPMN	-	+	+/-	The Oryx Editor [98]	+/- (Generic support with Rule Event)
BPEL4Chor	-	+	WS-BPEL [49]	The Oryx Editor [98]	-
Business Process Modeling Notation (BPMN)	+/-	+	WS-BPEL [49]	Microsoft Visio [83], Intalio STP BPMN Modeler [50], The Oryx Editor [98]	+/- (Generic support with Rule Event)

A detailed evaluation of business process modeling languages can be found in [63] [70].

2.4. Business rules

In this section, we describe basic business rules concepts, business rules categories, as well as a concrete business rule language, called REWERSE I1 Rule Markup Language (R2ML).

2.4.1. Business rules concepts

Business rule can be defined as “a statement that aims to influence or guide behavior and information in an organization” [125]. A rule also can be defined from two main perspectives, according to [140], from the information system perspective, where a business rule is a fact in a system stored as data and the constraint of values of such facts, and from the business perspective where a business rule includes some behavior of people in a system. From the business perspective, a business rule is defined as “*guidance that there is an obligation concerning conduct, action, practice, or procedure within a particular activity or sphere*” [140]. From the information system perspective, “*a business rule is a statement that defines or constrains some aspect of the business*” [140].

Business rules can be categorized according to their structure or source [125], as: Mandates (policies that must be followed, such as payment of taxes), Guidelines (rule that may or may not apply) and Policies (standards that should be applied to adhere some acceptable behavior).

There are different categories of rules, such as [142]:

- *Integrity rules* also known as (integrity) constraints consist of a constraint assertion, which is a sentence in a logical language such as first-order predicate logic or OCL. For example, the start date/time of a service must be later than the reservation date/time.
- *Derivation rules* where conditions resulting in conclusions. For example, a give 10% discount to a customer if he spends more than 1000$ per month.
- *Production rules* have a condition and a produced action, where condition is a logical formula. A produced action can execute an action. This rule type is important as it can be used to represent derivation or reaction rules. An example of the derivation rules is, if reservation date of a rental is 5 days in advance then give rental a 10 % discount.
- *Reaction rules* consist of a mandatory triggering event expression, an optional condition, and a produced action or a post-condition (or both). There are two types of reaction rules: those that do not have a post-condition, which are the well-known Event-Condition-Action (ECA) rules, and those that do have a postcondition, which we call ECAP rules. An example of reaction rule is, when an order is received, if a payment type is a “credit card” then ask for a credit card number.
- *Transformation rules* define change of state, for example, invoice type can be changed from “temporary” to “definitive”, but not opposite.

Business rules can be mainly characterized by the following elements [130]:

- Rules are important part of a business model.
- Rules are separated from business processes, not contained in them.
- Rules build of facts, while facts are expressed by terms. These terms “are used to express business concepts, while facts make assertions about these concepts, and rule constraints these facts.
- Rules should be defined by a business people.
- Ability to change rules is fundamental for business adaptability.
- Business is documented through rules.

Business rules are expressed by Business rule languages. Business rules can also be expressed in a natural language or in any formal language, such as: SWRL, R2ML, OCL, RuleML, etc. One type of a business rules are business rule markup languages. Business rule markup languages are usually based on some formal logic with expressive power [125]. They define what is required to take place, rather then how it should be accomplished. Some expect that rule markup languages will be the primary driving force for the widespread use of rules both on the Web and in distributed systems. They allow for deploying, executing, publishing and communicating rules on the Internet. They may also play the role of a lingua franca for exchanging rules between different systems and tools. They may be used, for example, to express derivation rules for enriching Web ontologies by adding definitions of derived concepts or for defining data access permissions; to describe and publish the reactive behavior of a system in the form of reaction rules; and to provide a complete XML-based specification of a software agent [143]. In a narrow sense, a rule markup language is concrete (XML-based) rule syntax. In a broader sense, it should have an abstract syntax as a common basis for defining various concrete languages serving different purposes. The main purpose of a rule markup language is to permit reuse, interchange and publication of rules.

2.4.2. Business rule languages

There are several rule languages and specification standards and they are mainly developed in two standardization streams. The first stream is lead by WWW consortium (W3C), which is responsible for Semantic Web and ontology languages, such as Rule Interchange Format (RIF) [38] and also there are Semantic Web Rule Language (SWRL) [46] and RuleML [12]. RIF is an official W3C Recomendation that defines a standard for sharing rules. That is, RIF is expressive enough to represent concepts of various rule languages, and it define that one should also develop a (two-way) transformation between RIF and any rule language that should be shared by using RIF. RIF include concrete XML serialization format for model-based languages such as RuleML, PRR and SBVR. Besides RuleML, the REWERSE I1 Rule Markup Language (R2ML) is well-known RIF implementation proposal. RuleML represents an initiative for creating a general rule markup language that should support different type of rules and different semantics. It is conceptualized to capture the hierarchy of rule types (reaction rules, derivation rules and integrity constraints). RuleML is built on logic programming paradigm of first order logic (i.e., predicate logic). In the tradition of logic programming that follows RuleML, research is focused on computable interpretations of predicate logic, by exploring a great number of semantic extensions and variations. SWRL is actually a combination of OWL and RuleML languages and it is used to reason over Semantic Web ontologies. This language is very similar to RuleML, and its rules are of the form of an implication between an antecedent (body) and a consequent (head). The intended meaning can be read as "whenever the conditions specified in the antecedent hold, then the conditions specified in the consequent must also hold". Both the antecedent (body) and consequent (head) consist of zero or more atoms. Multiple atoms are connected with the conjunction operator. We choosed to use R2ML language because it is also a rule modeling language, it does have XML concrete syntax and it can represent four main types of rules (Integrity, Derivation, Production and Reaction rules).

The second stream managed by Object Management Group (OMG), and it is primarily focused on object-oriented, business rules and process languages standards. Currently, there are three rule standardization efforts for Semantics of Business Vocabulary and Business Rules (SBVR) [95], Production Rule Representation [91] and Object Constraint Language [94]. SBVR include constructs for rule modeling vocabulary represented in semi-natural language. PRR is currently in finalization stage and it is designed for common high-level product rule representation. In UML, various model elements such as classes or state machines can be annotated by logical constraints defined by using OCL. In this way, UML models constrained by OCL expressions are more accurate and complete.

2.4.3. REWERSE I1 Rule Markup Language (R2ML) Language

Prof. Dr. Gerd Wagner and Dr. Adrian Giurca from the Institute of Informatics at Brandenburg University of Technology at Cottbus, Germany, have been working on the development of R2ML language prior to the version 0.2. From the version 0.2 to the actual version 0.5 (January 2009), Prof. Dr. Dragan Gašević from Athabasca University in Canada, and the author of this thesis have been involved in the R2ML development and implementation, as well.

The chosen approach for R2ML development is based the MDE principles [85]. The R2ML textual concrete syntax is defined in the form of an XML Schema. This schema is based on the R2ML MOF-based meta-model. From the perspective of the MDA, rules can be considered at three different abstraction levels (shown Figure 58).

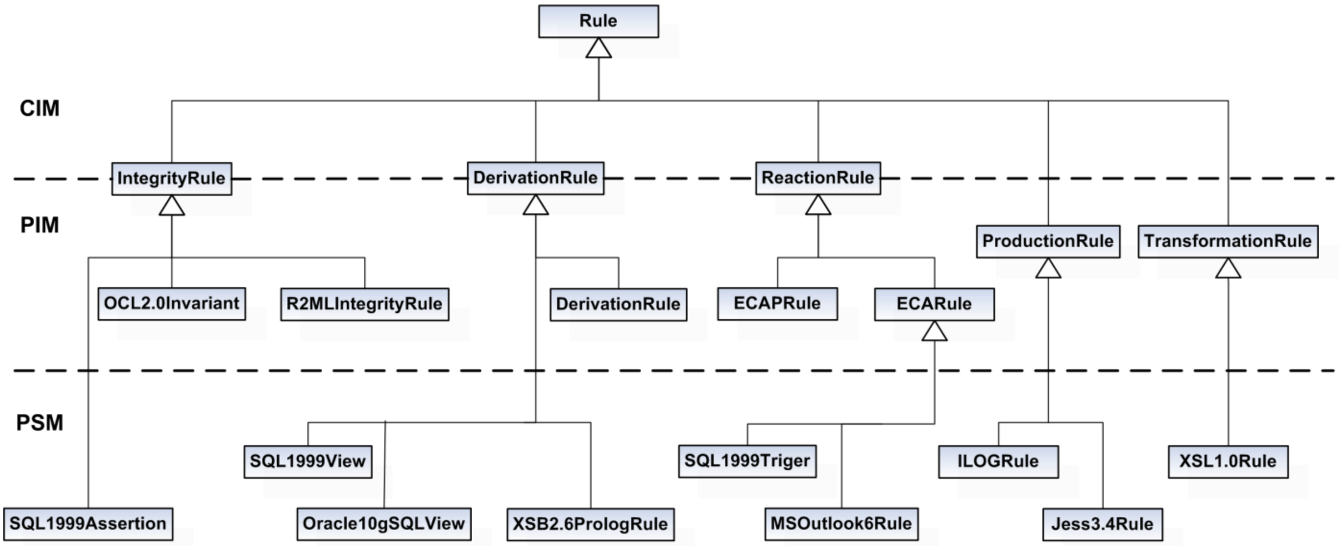


Figure 58. The concepts of rules at three different abstraction levels: computation independent (CIM), platform-independent (PIM) and platform-specific (PSM) modeling [142]

At the ('computation-independent') business domain level (CIM), rules are statements that express (certain parts of) a business/domain policy (e.g., defining terms of the domain language) in a declarative manner, typically using a natural language or a visual language. An example of such rule is: *The driver of a rental car must be at least 18 years old*. At the platform-independent operational design level (PIM), rules are formal statements, expressed in some formalism or computational paradigm, which can be directly mapped to executable statements of a software system. Examples of rule languages at this level are SQL:1999 [127] and OCL 2.0 [94]. At the platform-specific implementation level (PSM) rules are statements in a language of a specific execution environment, such as Oracle 10g views, Jess 3.4, XSB 2.6 Prolog or the Microsoft Outlook 6 Rule Wizard.

The R2ML language try to address all the requirements specified in RIF document given in [38] in order to provide a general markup language for sharing Web rules.

2.4.3.1. R2ML Metamodel and concrete XML-based syntax

The R2ML rule language is defined by a metamodel, by using the MOF metamodeling language [110] [142]. It can represent different rule constructs, i.e. it captures the most important types of rules. The actual version 0.5 (January 2009) supports the following types of rules: derivation rules, integrity rules (constraints), reaction rules, production rules and transformation rules (in early development).

In R2ML, rules are grouped into rule sets, which are also grouped into a rule base. Every rule base can have multiply vocabularies, such as RDFS, OWL or UML (see Figure 59).

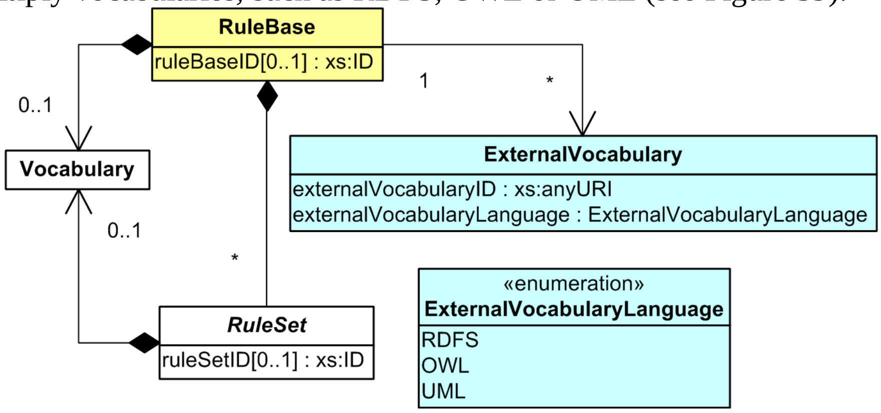


Figure 59. RuleBase in R2ML metamodel

Every type of rule in R2ML is contained in an appropriate rule set. Therefore, R2ML have rule sets for every rule type , as shown in Figure 60.

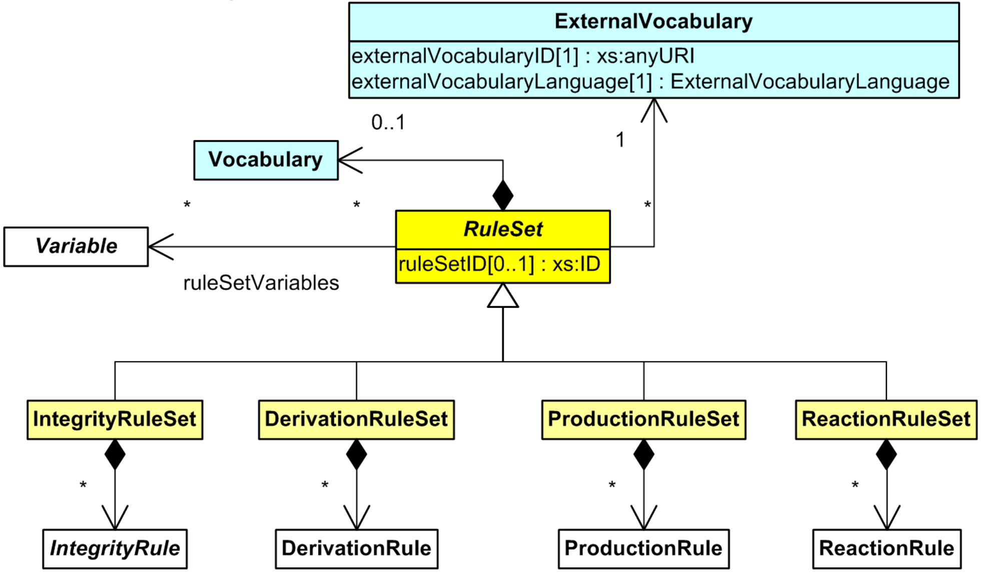


Figure 60. RuleSets in R2ML metamodel

2.4.3.1.1. *Integrity rules*

R2ML supports two kinds of integrity rules: *alethic* and *deontic* integrity rules (see Figure 61). An alethic integrity rule can be expressed with a phrase, such as "*it is necessarily the case that*", whereas a deontic one can be expressed with phrases, such as "*it is obligatory that*" or "*it should be the case that*" [143].

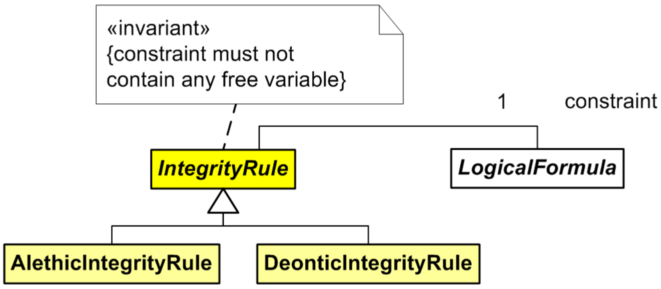
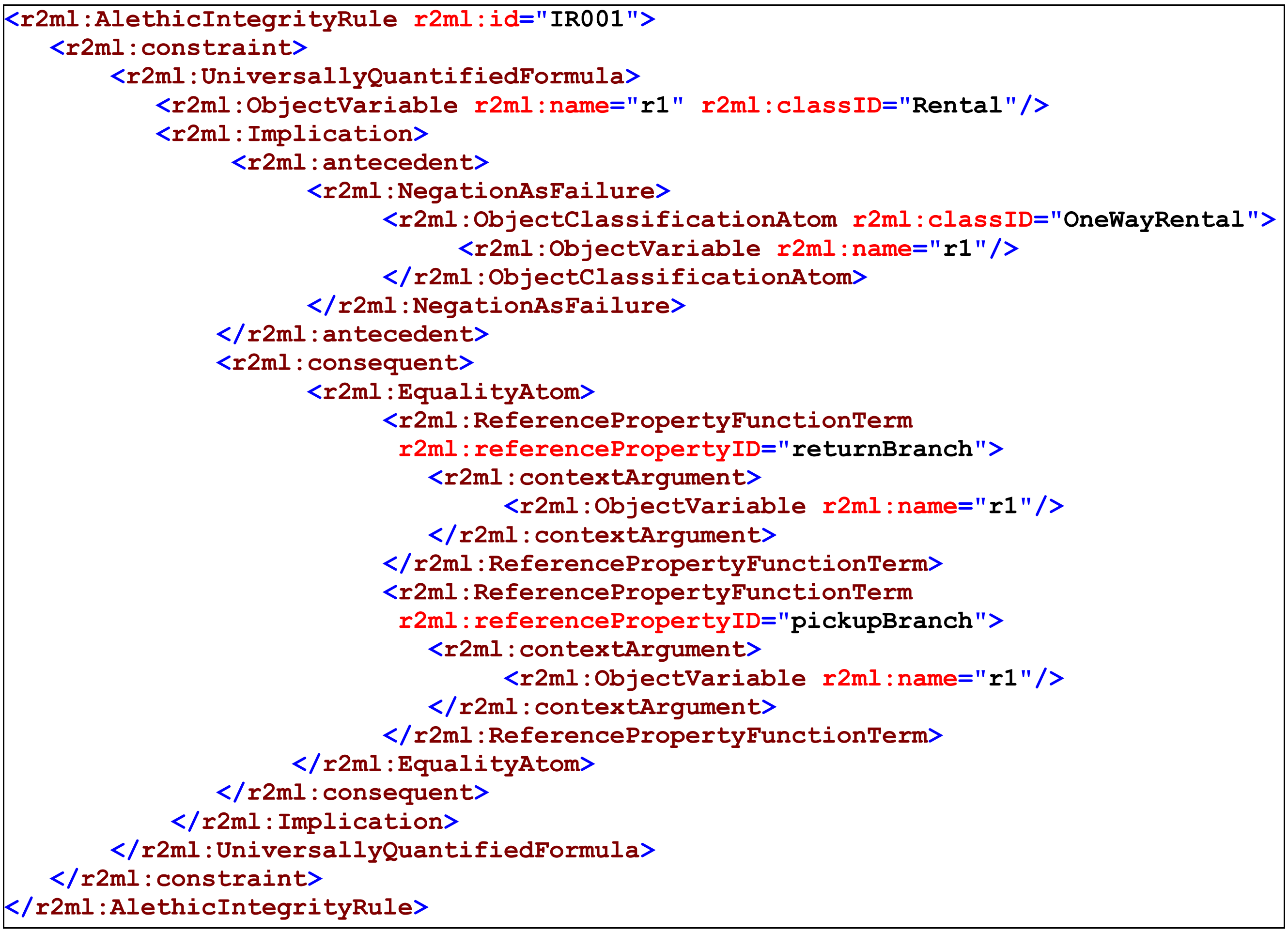


Figure 61. Integrity rules in the R2ML meta-model

A constraint assertion is a logical sentence that *must necessarily*, or that *should*, hold in all evolving states and state transition histories of the discrete dynamic system to which it applies. An integrity rule cannot have free variables, i.e. all variables from this formula are quantified. An example of integrity rule on CIM level is (from the EU-Rent case study[32]): *if rental is not a one way rental then return branch of rental must be the same as pick-up branch of rental*. This rule in R2ML XML concrete syntax is shown in Figure 62.



```
<r2ml:AlethicIntegrityRule r2ml:id="IR001">
   <r2ml:constraint>
       <r2ml:UniversallyQuantifiedFormula>
          <r2ml:ObjectVariable r2ml:name="r1" r2ml:classID="Rental"/>
          <r2ml:Implication>
               <r2ml:antecedent>
                    <r2ml:NegationAsFailure>
                         <r2ml:ObjectClassificationAtom r2ml:classID="OneWayRental">
                              <r2ml:ObjectVariable r2ml:name="r1"/>
                         </r2ml:ObjectClassificationAtom>
                    </r2ml:NegationAsFailure>
              </r2ml:antecedent>
              <r2ml:consequent>
                    <r2ml:EqualityAtom>
                         <r2ml:ReferencePropertyFunctionTerm
                          r2ml:referencePropertyID="returnBranch">
                            <r2ml:contextArgument>
                                 <r2ml:ObjectVariable r2ml:name="r1"/>
                            </r2ml:contextArgument>
                         </r2ml:ReferencePropertyFunctionTerm>
                         <r2ml:ReferencePropertyFunctionTerm
                          r2ml:referencePropertyID="pickupBranch">
                            <r2ml:contextArgument>
                                 <r2ml:ObjectVariable r2ml:name="r1"/>
                            </r2ml:contextArgument>
                         </r2ml:ReferencePropertyFunctionTerm>
                   </r2ml:EqualityAtom>
              </r2ml:consequent>
           </r2ml:Implication>
       </r2ml:UniversallyQuantifiedFormula>
   </r2ml:constraint>
</r2ml:AlethicIntegrityRule>
```

Figure 62. R2ML XML representation of an integrity rule

2.4.3.1.2. *Derivation rules*

Derivation Rules in R2ML have “conditions” and “conclusions” (see Figure 63), which are both logical formulas. In R2ML, the conditions of a derivation rule are `AndOrNafNegFormula`. Conclusions are restricted to a literal conjunction of atoms (see Figure 64).

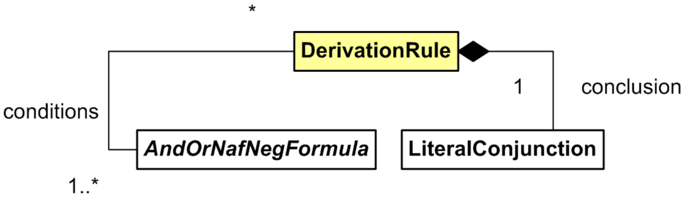


Figure 63. Derivation rules in the R2ML meta-model

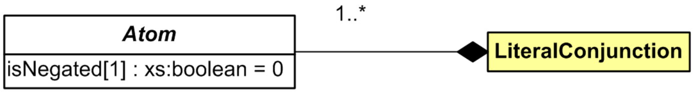
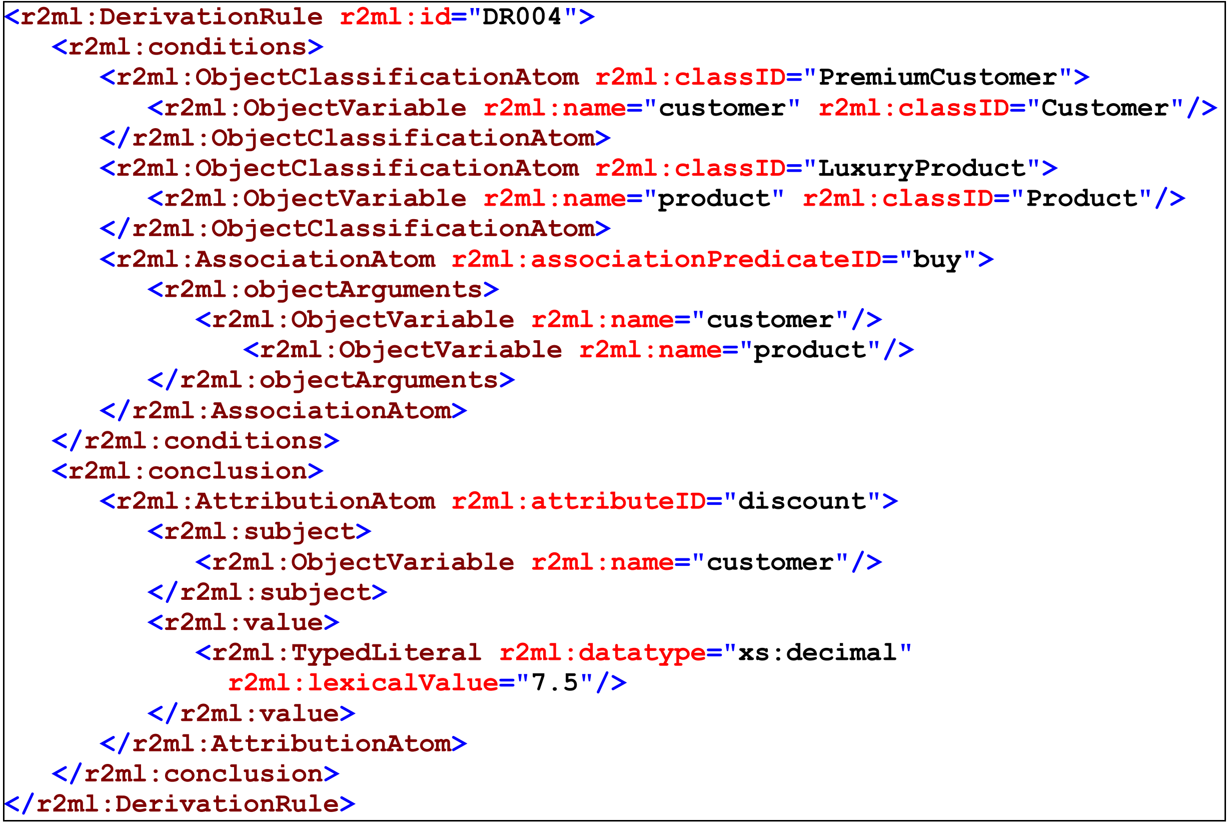


Figure 64. LiteralConjuction of R2ML atoms

An example of derivation rule on the CIM level is: *the discount for a customer buying a product is 7.5 percent if the customer is premium and the product is luxury*. This rule in the R2ML XML concrete syntax is shown in Figure 65.



```
<r2ml:DerivationRule r2ml:id="DR004">
   <r2ml:conditions>
      <r2ml:ObjectClassificationAtom r2ml:classID="PremiumCustomer">
         <r2ml:ObjectVariable r2ml:name="customer" r2ml:classID="Customer"/>
      </r2ml:ObjectClassificationAtom>
      <r2ml:ObjectClassificationAtom r2ml:classID="LuxuryProduct">
         <r2ml:ObjectVariable r2ml:name="product" r2ml:classID="Product"/>
      </r2ml:ObjectClassificationAtom>
      <r2ml:AssociationAtom r2ml:associationPredicateID="buy">
         <r2ml:objectArguments>
            <r2ml:ObjectVariable r2ml:name="customer"/>
               <r2ml:ObjectVariable r2ml:name="product"/>
         </r2ml:objectArguments>
      </r2ml:AssociationAtom>
   </r2ml:conditions>
   <r2ml:conclusion>
      <r2ml:AttributionAtom r2ml:attributeID="discount">
         <r2ml:subject>
            <r2ml:ObjectVariable r2ml:name="customer"/>
         </r2ml:subject>
         <r2ml:value>
            <r2ml:TypedLiteral r2ml:datatype="xs:decimal"
              r2ml:lexicalValue="7.5"/>
         </r2ml:value>
      </r2ml:AttributionAtom>
   </r2ml:conclusion>
</r2ml:DerivationRule>
```

Figure 65. R2ML XML representation of a derivation rule

2.4.3.1.3. Production rules

The conditions and post-conditions of production rules are `AndOrNafNegFormulas` (see Figure 66). A production rule may execute a `ProgramActionExpression`, as shown in Figure 88. While OCL can be used in a platform-independent production rule language to specify conditions on an object-oriented system state, the UML Action Semantics can be used to specify produced actions.

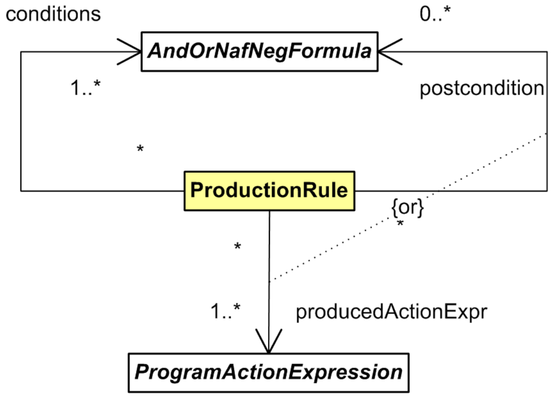
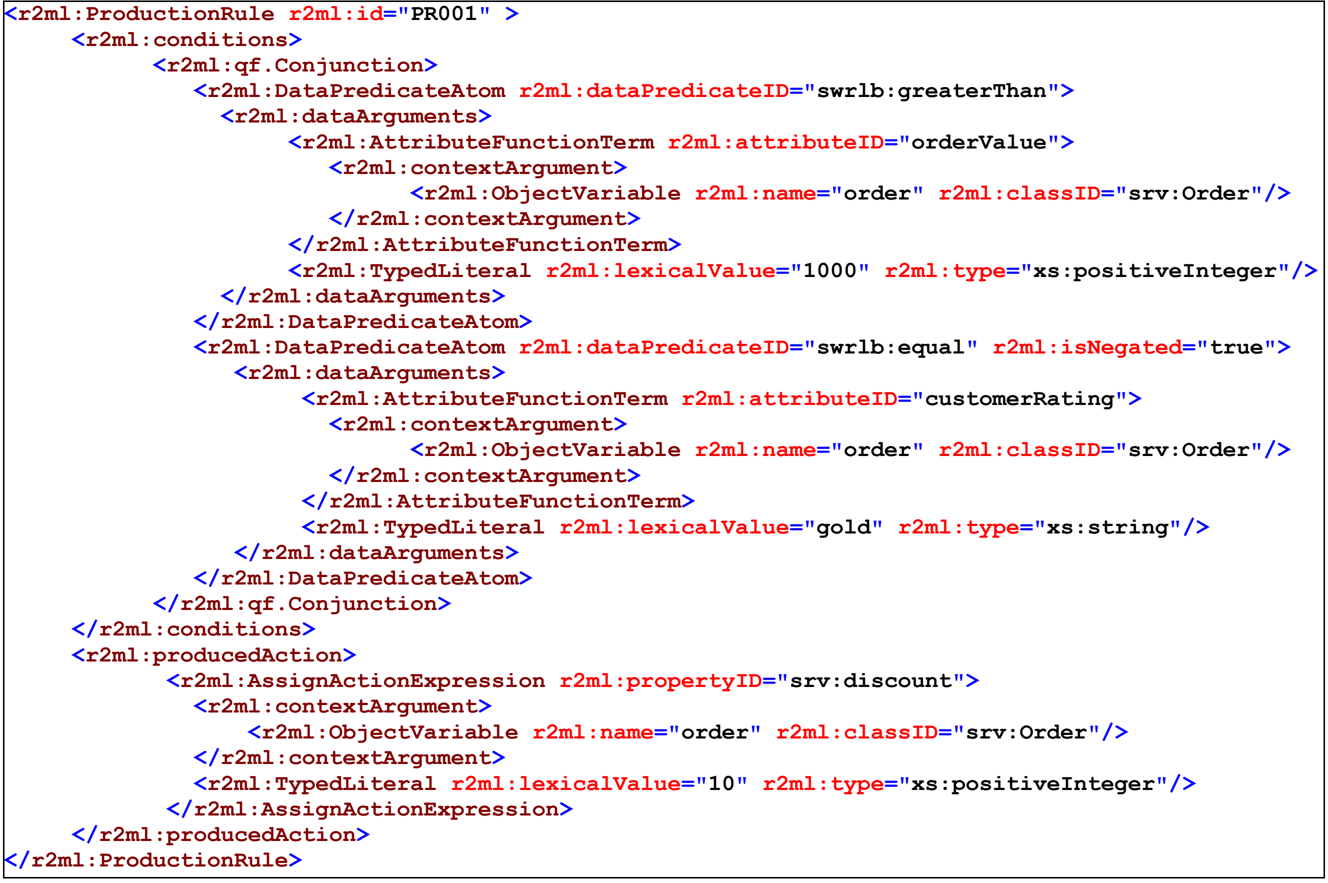


Figure 66. Production rules in the R2ML meta-model

An example of a production rule on the CIM level is: *if the order value is greater than 1000 and the customer type is not gold then give a 10% discount*. This rule in the R2ML XML concrete syntax is shown in Figure 67.



```
<r2ml:ProductionRule r2ml:id="PR001" >
     <r2ml:conditions>
           <r2ml:qf.Conjunction>
              <r2ml:DataPredicateAtom r2ml:dataPredicateID="swrlb:greaterThan">
                <r2ml:dataArguments>
                     <r2ml:AttributeFunctionTerm r2ml:attributeID="orderValue">
                        <r2ml:contextArgument>
                              <r2ml:ObjectVariable r2ml:name="order" r2ml:classID="srv:Order"/>
                        </r2ml:contextArgument>
                     </r2ml:AttributeFunctionTerm>
                     <r2ml:TypedLiteral r2ml:lexicalValue="1000" r2ml:type="xs:positiveInteger"/>
                </r2ml:dataArguments>
              </r2ml:DataPredicateAtom>
              <r2ml:DataPredicateAtom r2ml:dataPredicateID="swrlb:equal" r2ml:isNegated="true">
                 <r2ml:dataArguments>
                      <r2ml:AttributeFunctionTerm r2ml:attributeID="customerRating">
                        <r2ml:contextArgument>
                              <r2ml:ObjectVariable r2ml:name="order" r2ml:classID="srv:Order"/>
                        </r2ml:contextArgument>
                      </r2ml:AttributeFunctionTerm>
                      <r2ml:TypedLiteral r2ml:lexicalValue="gold" r2ml:type="xs:string"/>
                 </r2ml:dataArguments>
              </r2ml:DataPredicateAtom>
           </r2ml:qf.Conjunction>
     </r2ml:conditions>
     <r2ml:producedAction>
            <r2ml:AssignActionExpression r2ml:propertyID="srv:discount">
              <r2ml:contextArgument>
                  <r2ml:ObjectVariable r2ml:name="order" r2ml:classID="srv:Order"/>
              </r2ml:contextArgument>
              <r2ml:TypedLiteral r2ml:lexicalValue="10" r2ml:type="xs:positiveInteger"/>
            </r2ml:AssignActionExpression>
     </r2ml:producedAction>
</r2ml:ProductionRule>
```

Figure 67. R2ML XML representation of a production rule

2.4.3.1.4. Reaction rules

In R2ML, a reaction rules consist of a mandatory triggering event expression, an optional condition, and a triggered event expression or a post-condition (or both), which are roles of type `EventExpression`, `AndOrNafNegFormula` and `AndOrNafNegFormula`, respectively, as shown in Figure 68. While the condition of a reaction rule is exactly as the condition of a derivation rule, a quantifier-free formula, the post-condition is restricted to a conjunction of possibly negated atoms and it may be required to be satisfied.

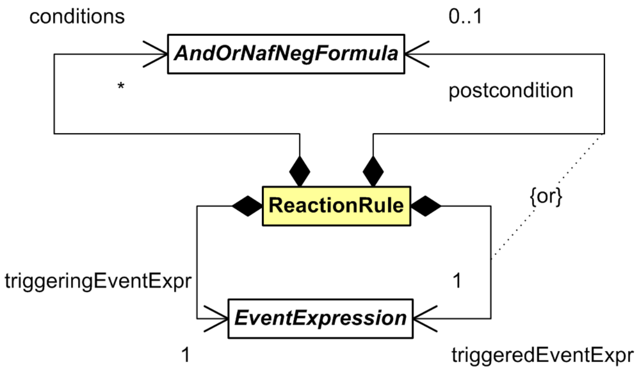


Figure 68. Reaction rules in the R2ML meta-model

A reaction rule consists of the following components:

- `triggeringEventExpr` and `triggeredEventExpr` is an R2ML `EventExpression`, which is either atomic or composite.
- `conditions` are represented as a collection of `AndOrNafNegFormula`, and as `postcondition`.

An example of reaction rule on CIM level is: *if customer returns a car and the car has more than 5000km from the last service then send the car to the service*. This rule in R2ML XML concrete syntax is shown in Figure 69.

```
<r2ml:ReactionRule r2ml:id="ECA001">
   <r2ml:triggeringEventExpr>
       <r2ml:MessageEventExpression r2ml:eventType="alert"
                                    r2ml:startTime="2006-03-21T09:00:00"
                                    r2ml:duration="P0Y0M0DT0H0M0S"
                                    r2ml:sender="http://www.mywebsite.org">
          <r2ml:arguments>
              <r2ml:ObjectVariable r2ml:name="car" r2ml:class="RentalCar"/>
               <r2ml:ObjectVariable r2ml:name="customer" r2ml:class="Customer"/>
          </r2ml:arguments>
       </r2ml:MessageEventExpression>
   </r2ml:triggeringEventExpr>
   <r2ml:conditions>
       <r2ml:DatatypePredicateAtom r2ml:datatypePredicate="ge">
          <r2ml:dataArguments>
             <r2ml:AttributeFunctionTerm r2ml:attribute="srv:lastservice">
                 <r2ml:contextArgument>
                         <r2ml:ObjectVariable r2ml:name="rentalCar"
                                              r2ml:class="srv:RentalCar"/>
                 </r2ml:contextArgument>
              </r2ml:AttributeFunctionTerm>
              <r2ml:AttributeFunctionTerm r2ml:attribute="odometer_reading">
                 <r2ml:contextArgument>
                        <r2ml:ObjectVariable r2ml:name="rentalCar"
                                              r2ml:class="srv:RentalCar"/>
                 </r2ml:contextArgument>
              </r2ml:AttributeFunctionTerm>
              <r2ml:TypedLiteral r2ml:datatype="xs:positiveInteger"
                                 r2ml:lexicalValue="5000"/>
          </r2ml:dataArguments>
       </r2ml:DatatypePredicateAtom>
    </r2ml:conditions>
    <r2ml:triggeredEventExpr>
       <r2ml:InvokeActionExpression r2ml:operation="service">
          <r2ml:contextArgument>
              <r2ml:ObjectVariable r2ml:name="rentalCar" r2ml:class="srv:RentalCar"/>
          </r2ml:contextArgument>
       </r2ml:InvokeActionExpression>
    </r2ml:triggeredEventExpr>
</r2ml:ReactionRule>
```

Figure 69. R2ML XML representation of a reaction rule

2.4.3.1.5. R2ML Vocabulary

R2ML language has a basic vocabulary that is defined to support basic rule constructs:

- Vocabulary for classification (basic): `Vocabulary`, `VocabularyEntry`, `Predicate`, `Property`, `Type`, `DatatypePredicate`, `Attribute`, `Class`, `Datatype` and `ObjectName`.
- Vocabulary for functional constructs[4]: `EnumerationDatatype`, `GenericFunction`, `DatatypeFunction`, `Operation`, `DataOperation` and `ObjectOperation`.
- Vocabulary for relational constructs[5]: `GenericPredicate` and `AssocationPredicate`.

Figure 70 shows the definition of elements for modeling vocabularies in the R2ML meta-model. All the previously described constructs are shown in this figure (blue classes are of the R2ML vocabulary and yellow classes are abstract classes).

[4] Functional constructs describe functional characteristics in a vocabulary; they represent operations that can be executed by some entity or can be used to translate some vocabulary elements (set) into other elements (set).

[5] Relational constructs defines relations between one or more vocabulary elements - if exists some relation between two vocabulary elements, then certain predicate will have *true* value. In opposite it will be *false*.

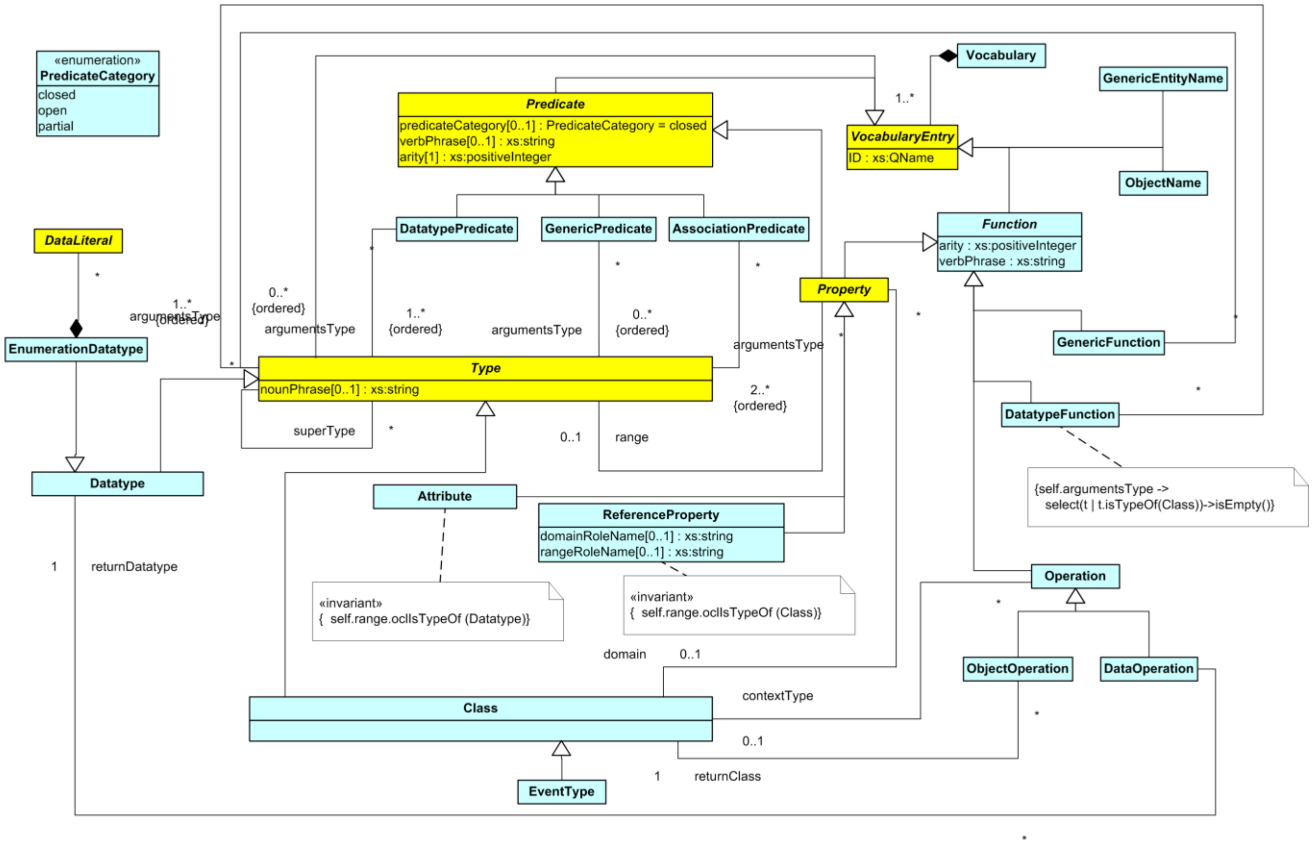


Figure 70. Vocabulary in the R2ML meta-model

2.4.3.1.5.1. Objects, Data, Variables

R2ML has three types of terms: `GenericTerm`, `ObjectTerm` and `DataTerm`. The concept of `ObjectTerm`, shown in Figure 71 is used for modeling variables that can be instantiated by object values and object constants. `ReferencePropertyFunctionTerm` is an object term that is used to model relations of a type similar to association ends in UML. `ObjectOperationTerm` is an object term that is used to model an operation on contextual argument.

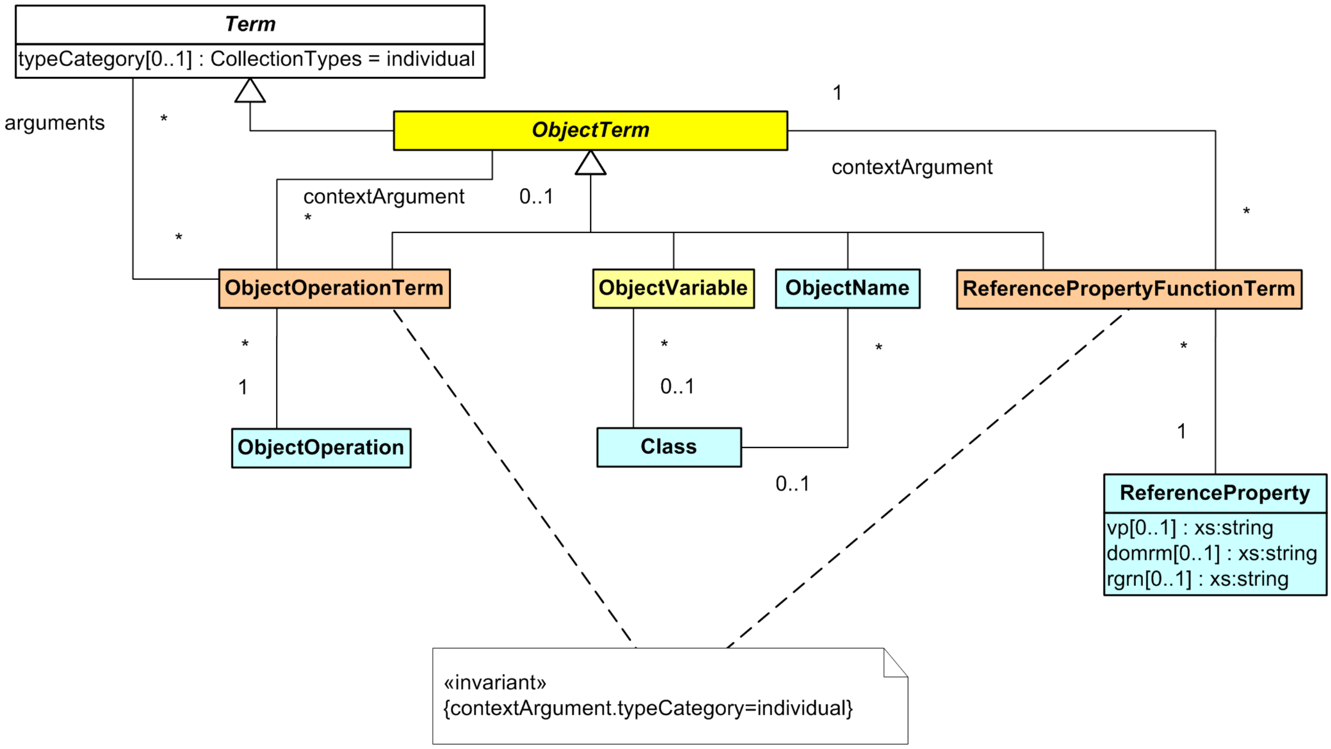


Figure 71. `ObjectTerm` in the R2ML meta-model

`DataTerm` is used to represent primitive data types and data values (see Figure 72). There are three types of data terms: `DataVariable`, which represents a variable, `DataLiteral`, which represents a value and `DataFunctionTerm`. `DataFunctionTerm` can be of three different types:

- `DatatypeFunctionTerm` represents arithmetic built-ins;
- `AttributeFunctionTerm` represents an attribute function (a function, which returns attribute value for an object);
- `DataOperationTerm` represents a user-defined function (method, for instance) and takes `DataTerm` or `ObjectTerm` as parameters.

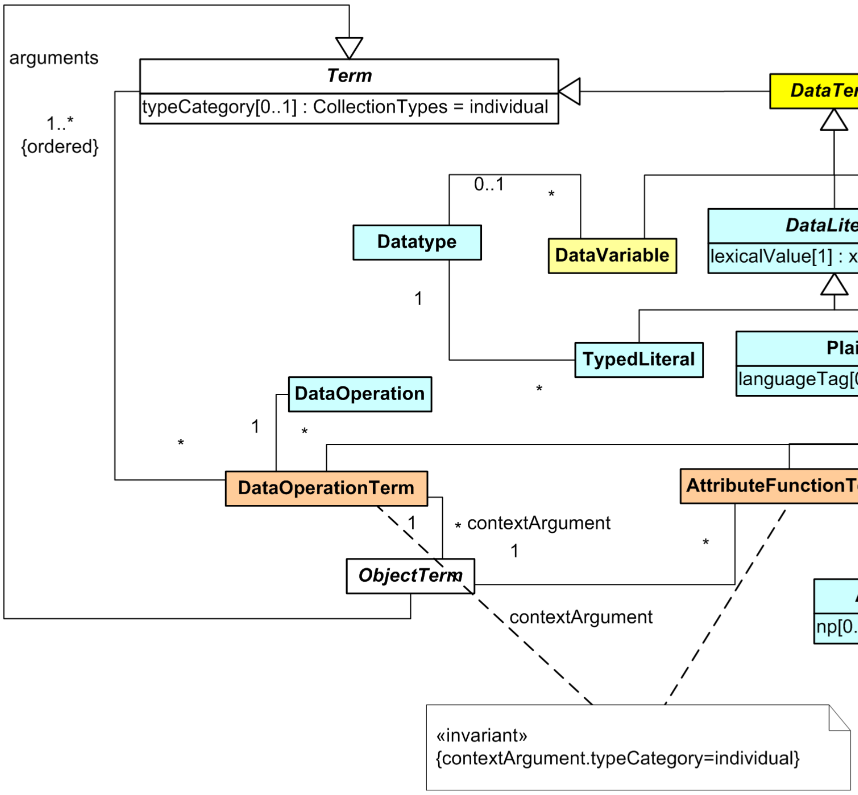


Figure 72. `DataTerm` in the R2ML meta-model

A `GenericTerm` (see Figure 73) is used for modeling variables that may, or may not, have a data type (`GenericVariable`), constants (`GenericEntityName`), and generic functions (`GenericFunctionTerm`).

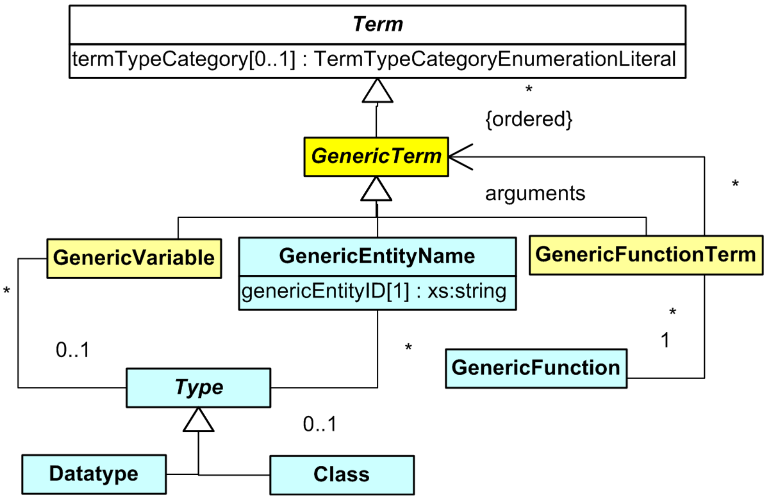


Figure 73. `GenericTerm` in the R2ML meta-model

R2ML supports three types of variables: `GenericVariable`, `ObjectVariable` and `DataVariable` (see Figure 74).

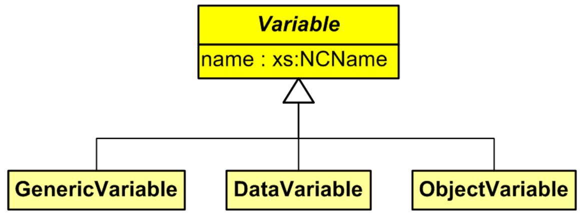


Figure 74. Variables in the R2ML meta-model

2.4.3.1.5.2. Atoms

The basic constituent of a rule is an atom. R2ML defines a meta-model for atoms. The R2ML atoms are compatible with all important concepts of OWL, SWRL and RuleML. All atoms from the R2ML meta-model are presented on Figure 75.

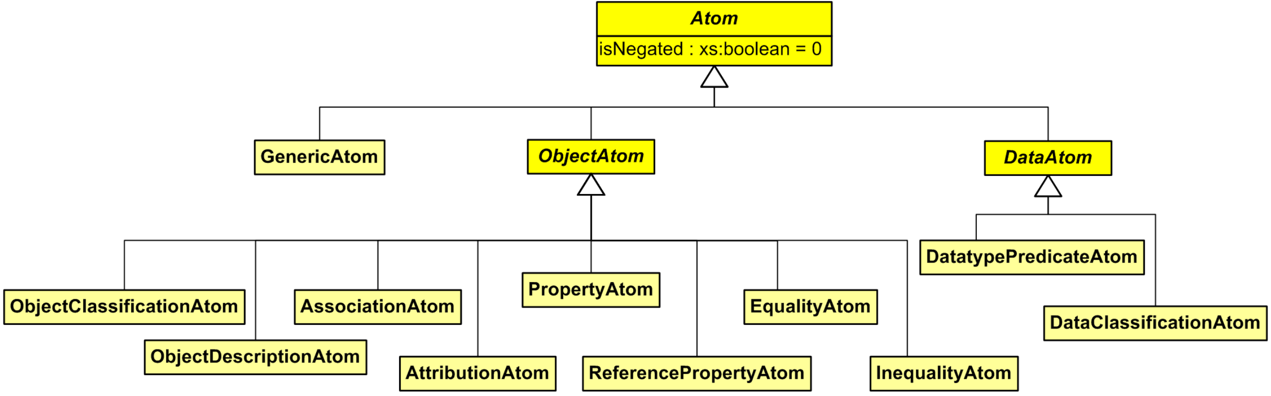


Figure 75. Atoms in the R2ML meta-model

An object classification atom (see Figure 76) refers to a class and consists of an object term.

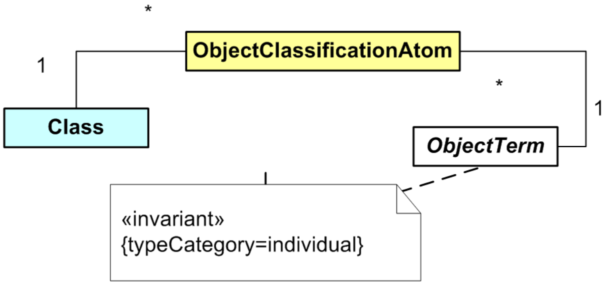


Figure 76. `ObjectClassificationAtom` in the R2ML meta-model

An object description atom (see Figure 77) refers to a class as a base type, and to zero or more classes as categories. It consists of a number of slots (attribute data slot and reference property object slot). An instance of such atom refers to one particular object.

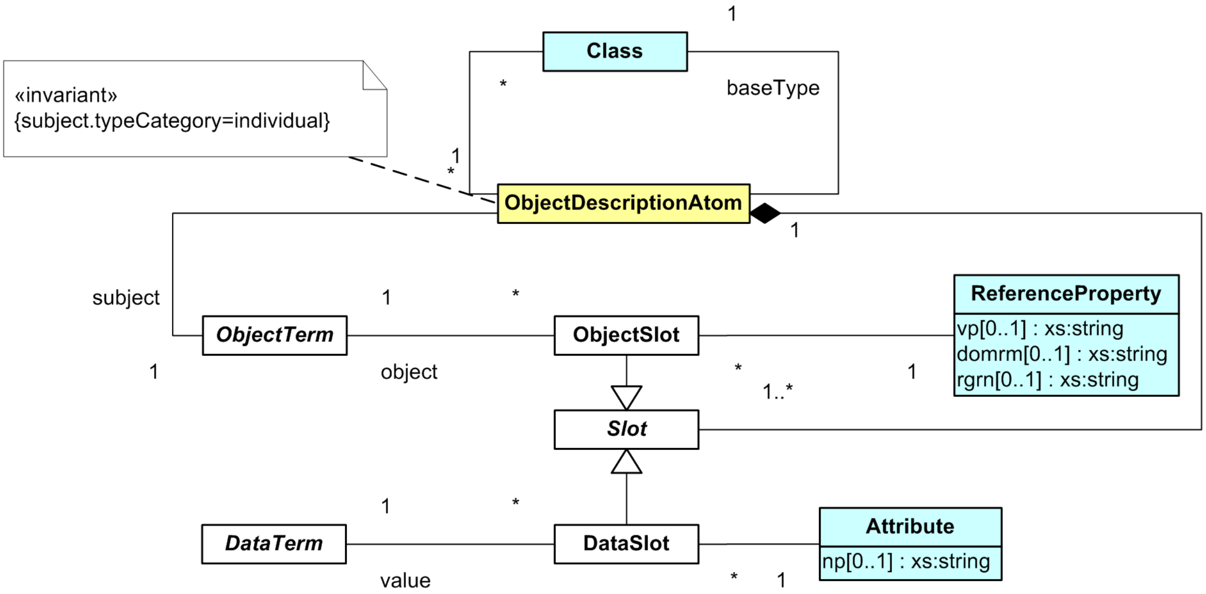


Figure 77. `ObjectDescriptionAtom` in the R2ML meta-model

An attribution atom (see Figure 78) consists of an object term as ”subject”, and a data term as ”value”.

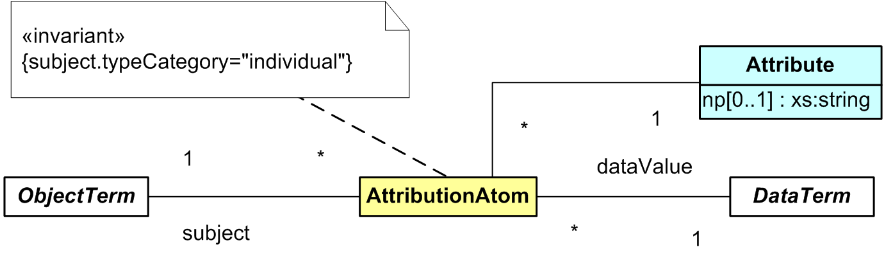


Figure 78. `AttributionAtom` in the R2ML meta-model

A reference property atom (see Figure 79) associates object terms as ”subjects” with other object terms as ”objects”. In order to directly support common fact types of natural language, it is important to have n-ary predicates (for n > 2).

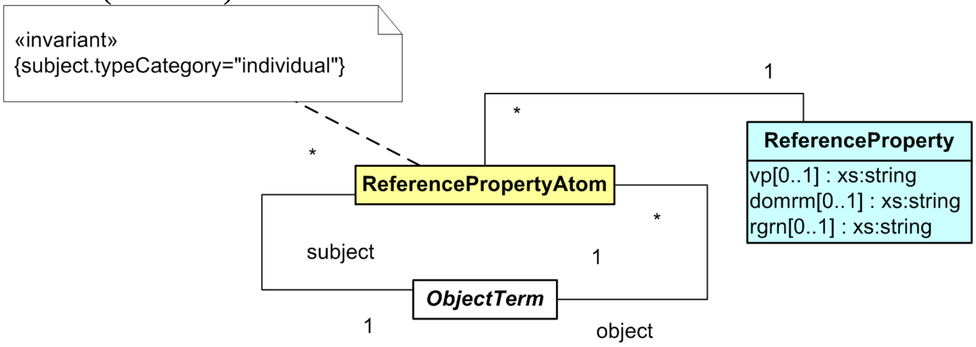


Figure 79. `ReferencePropertyAtom` in the R2ML meta-model

An association atom (see Figure 80) is constructed by using an n-ary predicate as an association predicate, a collection of data terms as “data arguments”, and a collection of object terms as “object arguments”.

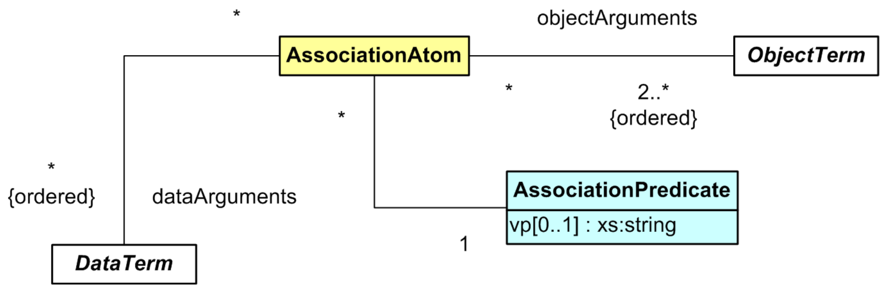


Figure 80. `AssociationAtom` in the R2ML meta-model

Both equality atom and inequality atom (see Figure 81) are composed of two or more object terms.

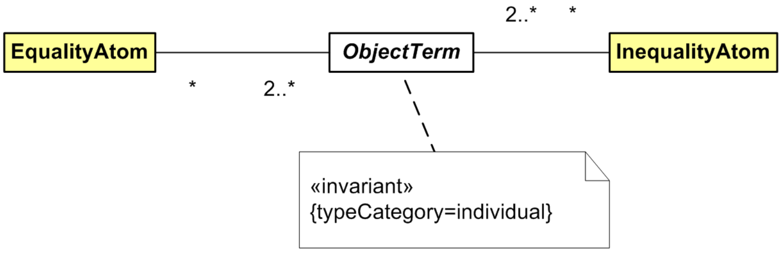


Figure 81. `EqualityAtom` and `InequalityAtom` in the R2ML meta-model

A data classification atom (see Figure 82) consists of a data term and refers to a data type.

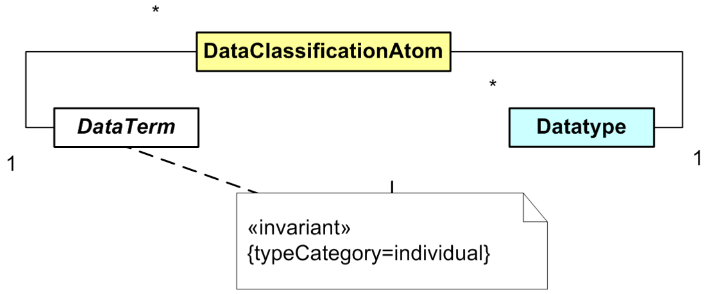


Figure 82. `DataClassificationAtom` in the R2ML meta-model

R2ML metamodel also include concept of datatype predicate atom, which refers to a data term and datatype predicate (see Figure 83).

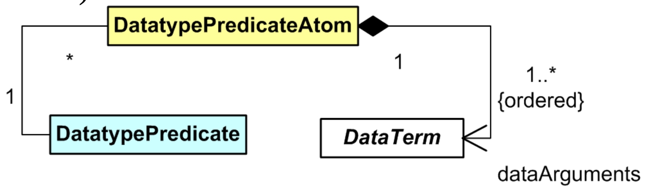


Figure 83. `DatatypePredicateAtom` in the R2ML meta-model

`GenericAtom` consists of a predicate (which can be of type: `ObjectClassificationPredicate`, `AttributionPredicate`, `AssociationPredicate`, `ReferencePropertyPredicate`, `EqualityPredicate`, `InequalityPredicate`, `DatatypePredicate` and `DataClassificationPredicate`) and arguments, as shown in Figure 84.

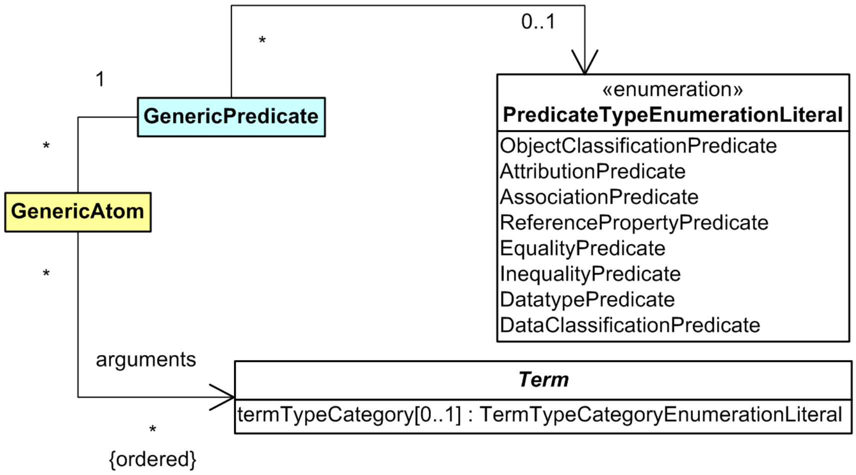

0..1
*
1
GenericPredicate
*
GenericAtom
*
«enumeration»
PredicateTypeEnumerationLiteral
ObjectClassificationPredicate
AttributionPredicate
AssociationPredicate
ReferencePropertyPredicate
EqualityPredicate
InequalityPredicate
DatatypePredicate
DataClassificationPredicate
arguments
Term
termTypeCategory[0..1] : TermTypeCategoryEnumerationLiteral
*
{ordered}

Figure 84. `GenericAtom` in the R2ML meta-model

2.4.3.1.5.3. Formulas

R2ML provides two abstract concepts for formulas: the concept of `AndOrNafNegFormula` (see Figure 85), which represents the most general quantifier-free logical formula with weak and strong negations, and the concept of `LogicalFormula` (see Figure 86), which corresponds to a general first order formula.

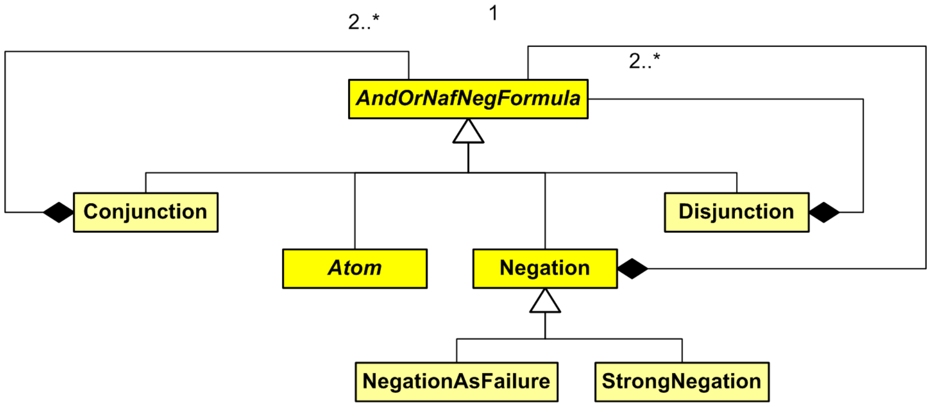


Figure 85. `AndOrNafNegFormula` in the R2ML meta-model

R2ML supports two kinds of negation (as shown in Figure 86). The distinction between weak and strong negation is used in several computational languages (like SQL [127] and OCL [94]), and it is presented in [141]. A weak negation captures the absence of positive information, while a strong negation captures the presence of explicit negative information. The weak negation captures the computational concept of negation-as-failure (or closed-world negation).

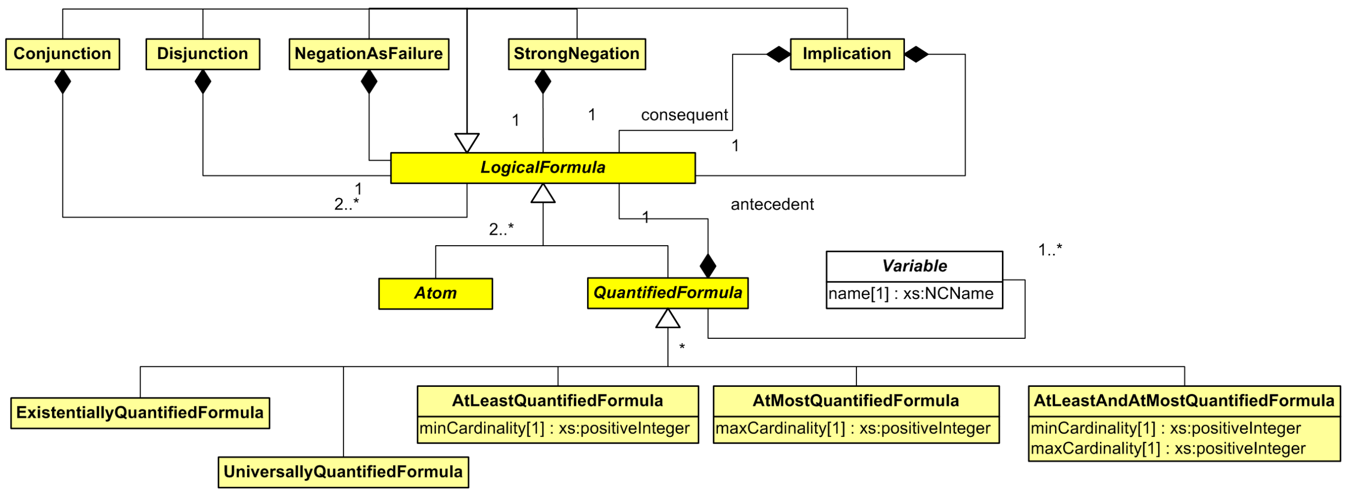

Conjunction
Disjunction
NegationAsFailure
StrongNegation
Implication
consequent
1
1
1
LogicalFormula
1
2..*
2..*
antecedent
1
Atom
QuantifiedFormula
Variable
name[1] : xs:NCName
1..*
*
ExistentiallyQuantifiedFormula
UniversallyQuantifiedFormula
AtLeastQuantifiedFormula
minCardinality[1] : xs:positiveInteger
AtMostQuantifiedFormula
maxCardinality[1] : xs:positiveInteger
AtLeastAndAtMostQuantifiedFormula
minCardinality[1] : xs:positiveInteger
maxCardinality[1] : xs:positiveInteger

Figure 86. `LogicalFormula` in the R2ML meta-model

2.4.3.1.5.4. Actions

R2ML `EventExpression` concept include following elements: `AtomicEventExpression`, `AndNotEventExpression`, `SequenceEventExpression`, `ParallelEventExpression` and `ChoiceEventExpression` (see Figure 87). These elements are used to represent composite actions, like sequential or parallel actions, while `AndNotEventExpression` contains two `EventExpression`'s A, B such that "A occurs and B does not."

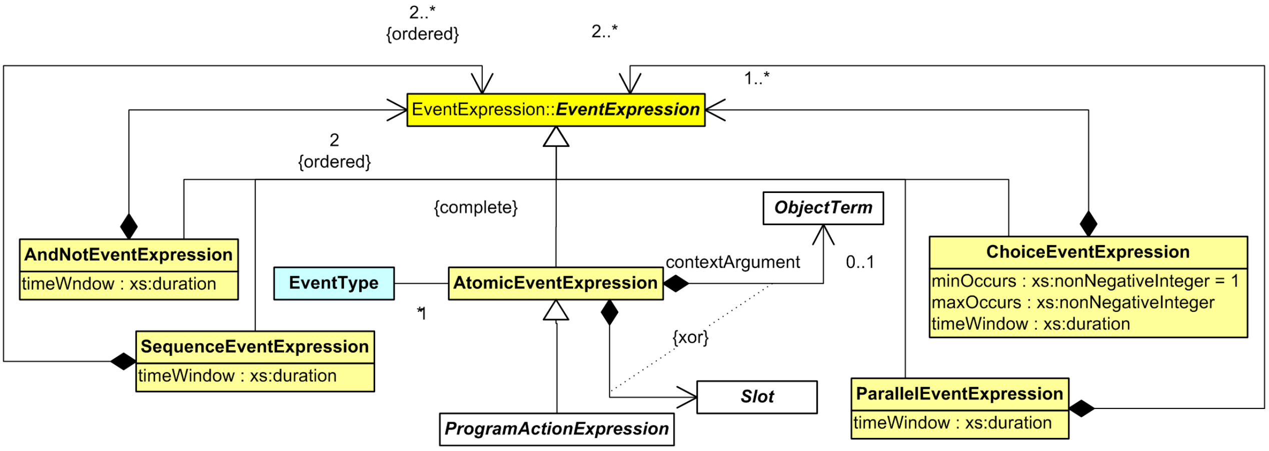


Figure 87. Event expressions in the R2ML meta-model

R2ML supports five types of actions (`ProgramActionExpression`, corresponds to the OMG PRR actions [91]): `InvokeAction`, `AssignAction`, `CreateAction`, `DeleteAction` and `SOAPAction` (see Figure 88). These actions create state changes in the rule system working memory (i.e., change facts in the working memory).

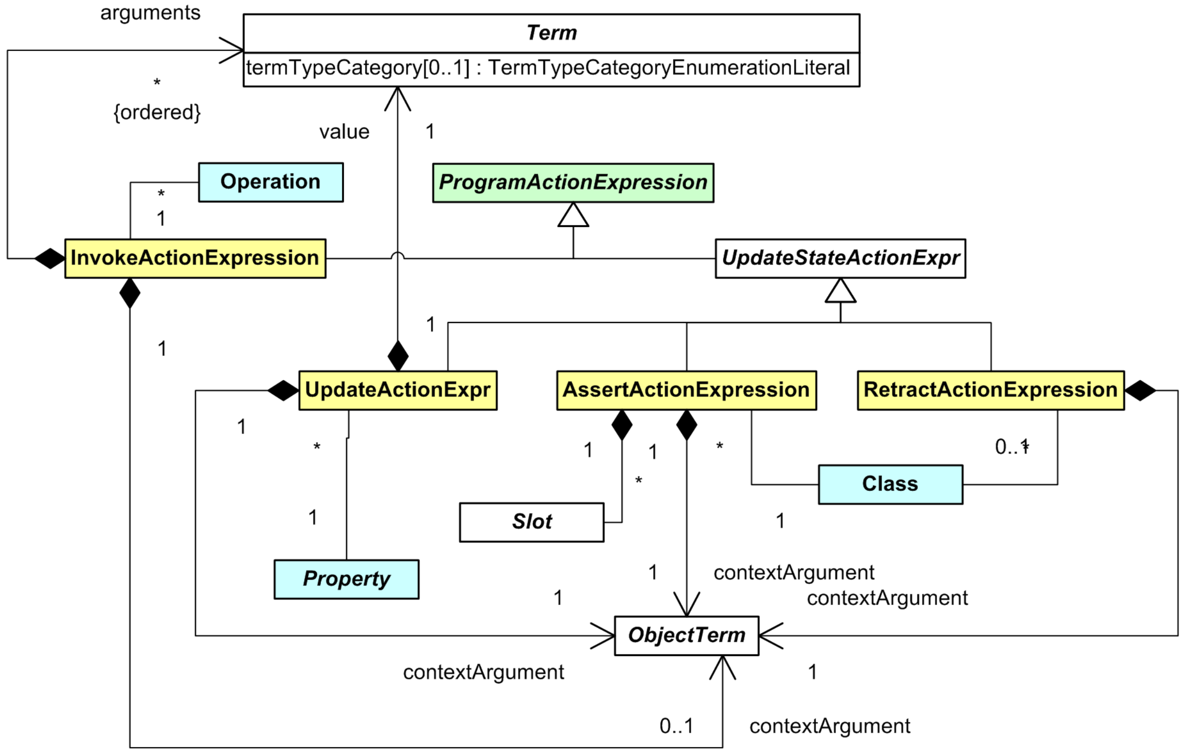


Figure 88. Actions in the R2ML meta-model

- InvokeActionExpression refers to an UML Operation and contains a list of arguments. This action invokes an operation with a list of arguments.
- UpdateActionExpr refers to an UML Property and contains a DataTerm. This action assigns a value to a property.
- AssertActionExpression refers to an UML Class and to a Slot as a parameter. Assert action is used to create a new instance in working memory.
- RetractAction refers to an UML Class and this action is used to remove instances from working memory.

As shown in Figure 87, the AtomicEventExpression can have an event type. These event types can be time or message-related, as shown in Figure 89.

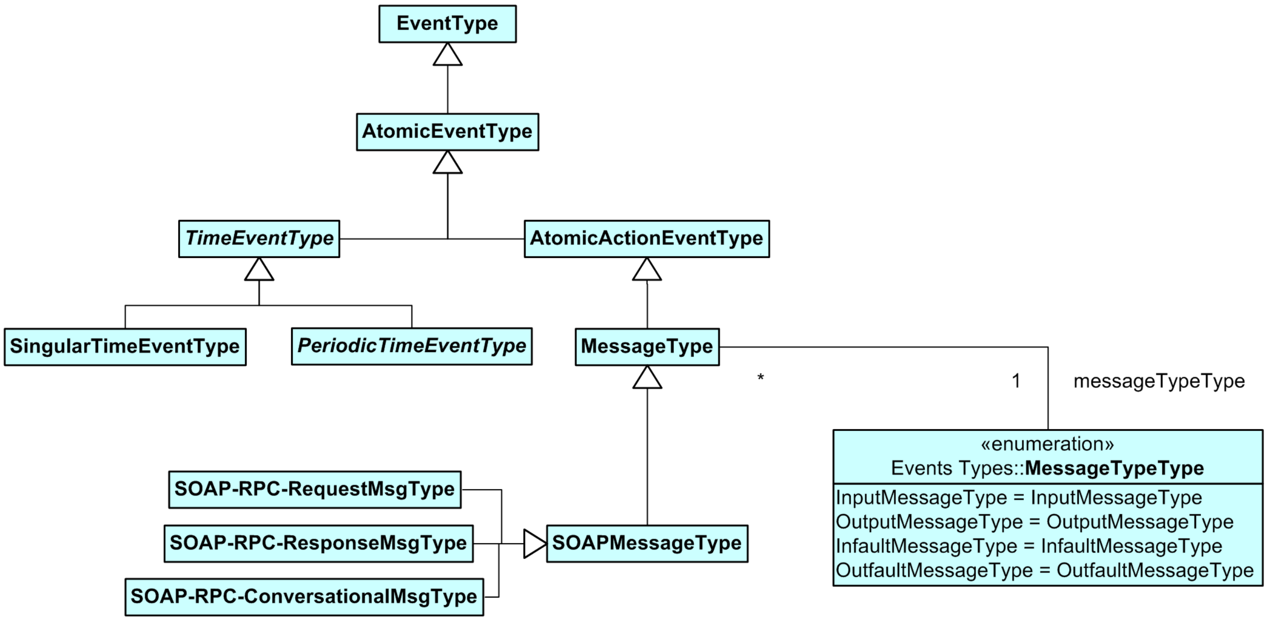

EventType
AtomicEventType
TimeEventType
AtomicActionEventType
SingularTimeEventType
PeriodicTimeEventType
MessageType
*
1
messageTypeType
«enumeration»
Events Types::MessageTypeType
InputMessageType = InputMessageType
OutputMessageType = OutputMessageType
InfaultMessageType = InfaultMessageType
OutfaultMessageType = OutfaultMessageType
SOAP-RPC-RequestMsgType
SOAP-RPC-ResponseMsgType
SOAP-RPC-ConversationalMsgType
SOAPMessageType

Figure 89. Vocabulary events in the R2ML metamodel

2.4.3.2. UML-Based Rule Modeling Language (URML)

UML-Based Rule Modeling Language (URML) represents one possible implmentation of the R2ML graphical notation [72]. URML is developed as an extension of the UML metamodel to be used for rule modeling. The name is defined in such a way, as the URML graphical notation is defined in such a way to remind completely of the graphical notation of the UML class models; just with an addition that rules are defined on top of such models. A previous implementation of URML in the Strelka tool [73], implemented as both Fujaba and Eclipse plug-ins, is using a different implementation than it is followed in the implementation presented in this theses later on. Namely, Strelka is designed as a heavyweight UML profile through the extension of the UML2 metamodel, while in this thesis we will introduce R2ML graphical syntax as an implementation of the R2ML metamodel presented in the previous section (see section 3.1).

2.4.3.2.1. URML graphical notation

All types of rules in URML are depicted using circles with identifiers, except integrity rules. Integrity rules are represented as OCL invariants on URML vocabulary models. Conditions are depicted as arrows from a conditioned model element to a rule circle (i.e., →). A condition element can be one of the UML classifiers: *class, association,* and *association end.* A negated condition is depicted using a crossed arrow (i.e., ↛). Conditions can also be defined by using OCL filters, that is, OCL expressions that further constrain the conditions defined by means of UML classifiers (e.g., price = cvar.wantedPrice). A post-condition is depicted as an outgoing arrow with a double head (↠, in order to denote a state change) from the rule circle to the post-condition classifier (class, association or association end). An instance of the *EventCondition* class is depicted as an incoming arrow (—►) from the message type or fault message type class to the rule circle. An event condition refers to an object variable, which represents an instance of the message event class (<<message event type>>) and corresponds to a annotation of a event condition arrow with the variable name. A rule action has a message event type as an output message or fault message type as an out-fault. In the URML visual notation, the *RuleAction* class is depicted as a double headed outgoing arrow (↠) from the rule circle to the message type class, fault type class or action. A rule action refers to an object variable, which represents an instance of the *MessageEventType* class and corresponds to the annotation of the action arrow with the variable name. A rule action arrow is annotated with an action type (A for the *Assert* action, R for the *Retract* action, U for the *Update* action, and I for the *Invoke* action).

We show elements of the URML syntax in Table IV.

Table IV. URML's concrete graphical notation

Stereotype	**UML 2.0 metaclass**	**Graphical notation**
DerivationRule	Class	DR
ProductionRule	Class	PR
ReactionRule	Class	RR
RuleCondition	Class, Association, AssociationEnd	→
Negated Condition	Class, Association, AssociationEnd	⇸
PostCondition	Class, Association, AssociationEnd	↠
EventCondition	Class, Association, AssociationEnd	➔
RuleAction	Class, Association, AssociationEnd	↠

2.4.3.2.2. Modeling reaction rules

Reaction rules in URML are modeled by an atomic triggering event, denoted by a class with <<event> stereotype. In Figure 90, we show an example of a URML reaction rule. This reaction rule, is triggered by a message inquiring about the price of flights, and returns a message that carries the price information if a certain flight price is equal to the wanted flight price. The URML class that represents the input message (*CheckPriceRequest*) of the reaction rule is an event depicted with the <<event>> stereotype on UML class. The same stereotype is also the type of the reaction rule output message (*CheckPriceResponse*). The input message *CheckPriceRequest* is connected with a *Class* instance type called *Airline*, by using an association. The condition is represented by a *Flight* class that connects with RR and the condition expression defined on this connection (*price = cvar.wantedPrice*).

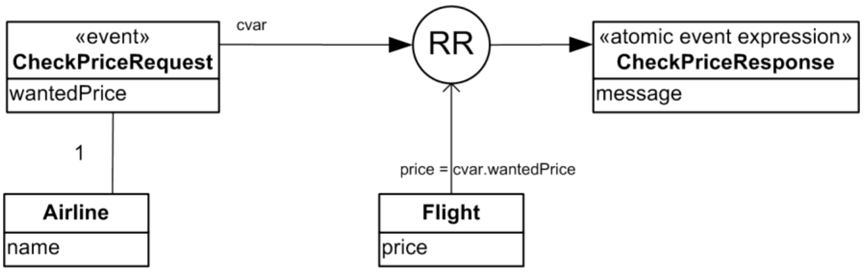


Figure 90. Reaction rule modeling

We should note that currently composite events are not supported in URML as it is complex to recognize such events in rule engine implementations.

In our previous work [45][71], we presented how reaction rules can be translation into Web services, i.e., WSDL descriptions [71]. We have done this in the following way. A triggering event of a

RR maps to the input message of a Web service operation. The action of the RR, which is triggered when a condition is true, maps to the output message of the Web service operation. To model condition constructs (e.g., *price = wantedPrice*) we use OCL filters [94]. OCL filters are based on a part of OCL that models logical expressions, which can be later translated to R2ML logical formulas, as parts of reaction rules. However, these OCL filters cannot be later translated to Web service descriptions (e.g., WSDL), as those languages cannot support such constructs. This means that for each Web service, we can generate a complementary rule, which fully regulates how its attributed service is used.

2.4.3.2.3. Modeling derivation rules

An example of a derivation rule modeled in URML is shown in Figure 91. This rule checks if if a rental car is scheduled for service. This condition for a care to be scheduled is if the last maintenance date was more than 90 days or the service reading greater than 5000. The condition for this rule is defined as an OCL filter [94].

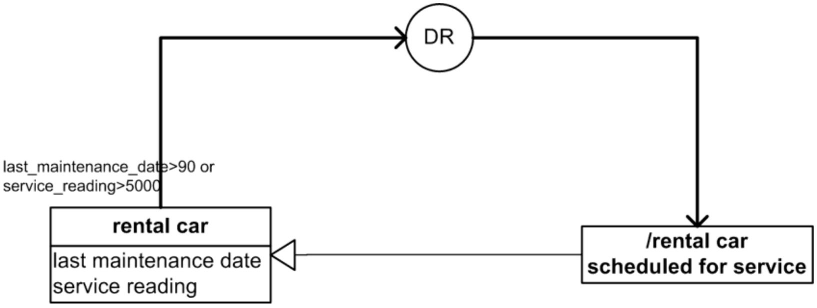


Figure 91. Derivation rule modeling

2.4.3.2.4. Modeling production rules

Figure 92 show an example of a production rule modeled in URML. This rule has a simple condition that states when an order value is greater then 1000 then given discount is 6, however if this condition is less then 1000 then given discount is 3. An assert action, depicted with “A” character is used to assign a new value to the “discount” variable.

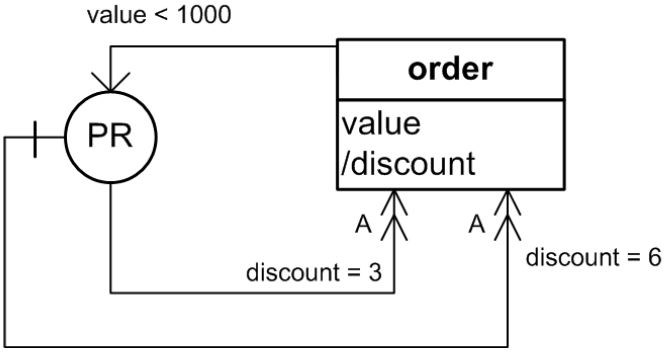


Figure 92. Production rule modeling

2.4.3.3. Policy Modeling Language (PML)

Policies are used to “regulate the behavior of system components without changing code and without requiring the consent or cooperation of the components being governed” [20]. Policies are usually used to constrain the behaviour of a system. Polcies are usually represented in policy definition languages, such as Rei, KAoS, Ponder, PML, etc. In this thesis use PML, as it represents an extension of R2ML that specialize R2ML’s logical foundation, metamodel, both graphical and XML-based textual concrete syntax, and transformations with the policy-specific concepts [56]. This language represents a policy modeling language by abstracting common policy concepts from several policy languages and by grounding it on the sound theoretical foundation of deontic logic. The language can be used in the software analysis and design phases together with other well-established languages (e.g., UML) and can be deployed (implementation and integration phases) and transformed to different policy languages and used with different technologies (e.g., for business vocabularies and rules, components, and processes), such as KaOS and Rei by using QVT-based transformations [56].

PML is also concerned with the very many existing policy languages, each of them proposed with the goal of protecting the privacy of information and authorizing requesters by providing different levels of access to the available resources and information. Each policy language is based on a particular type of logic (ranging from First Order Logic – FOL – to its subsets, i.e., description or computational logic, and their specialized variants, e.g., deontic logic [6][44].

2.4.3.3.1. Policy Modeling Language Metamodel

Policy Definition Metamodel is a MOF-based definition of PML. The PML abstract syntax is defined by extending the MOF-based metamodel of R2ML. Figure 93 shows the metamodel for PML. Following the concepts of deontic logic, PML provides support for defining *permission*, *prohibition*, *obligation*, and *dispensation* policy rules. It also provides a conceptualization of *Actions* that the policies are defined over, *Actors* for these actions, and the *Context* to which these policies are applied. All these classes are derived from the super class *Entity* which resembles the *owl:Thing* element of the OWL language. It also should be noted that, following the MOF regulations and rules, all these classes could be extended to define more fine-grained policies.

In Figure 93, we also show the icons used to graphically represent PML constructs in the General Policy UML profile. As shown in the figure, a policy rule is defined as an R2ML derivation rule with its condition part composed of a conjunction of logical formulas and its conclusion part containing an R2ML `ObjectDescriptionAtom`. For each policy rule, this element holds the description for an instance of one of the modal concepts in the deontic logic, i.e., permission, prohibition, obligation, or dispensation. The policy instance represented by an `ObjectDescriptionAtom` conveys the derived policy decision upon satisfaction of the logical formulas in the condition.

[66] One of the main groups for modal logic that is concerned with what we ought to do, what we are allowed to do, and must not do, aka, obligatory, permitted, and forbidden acts.

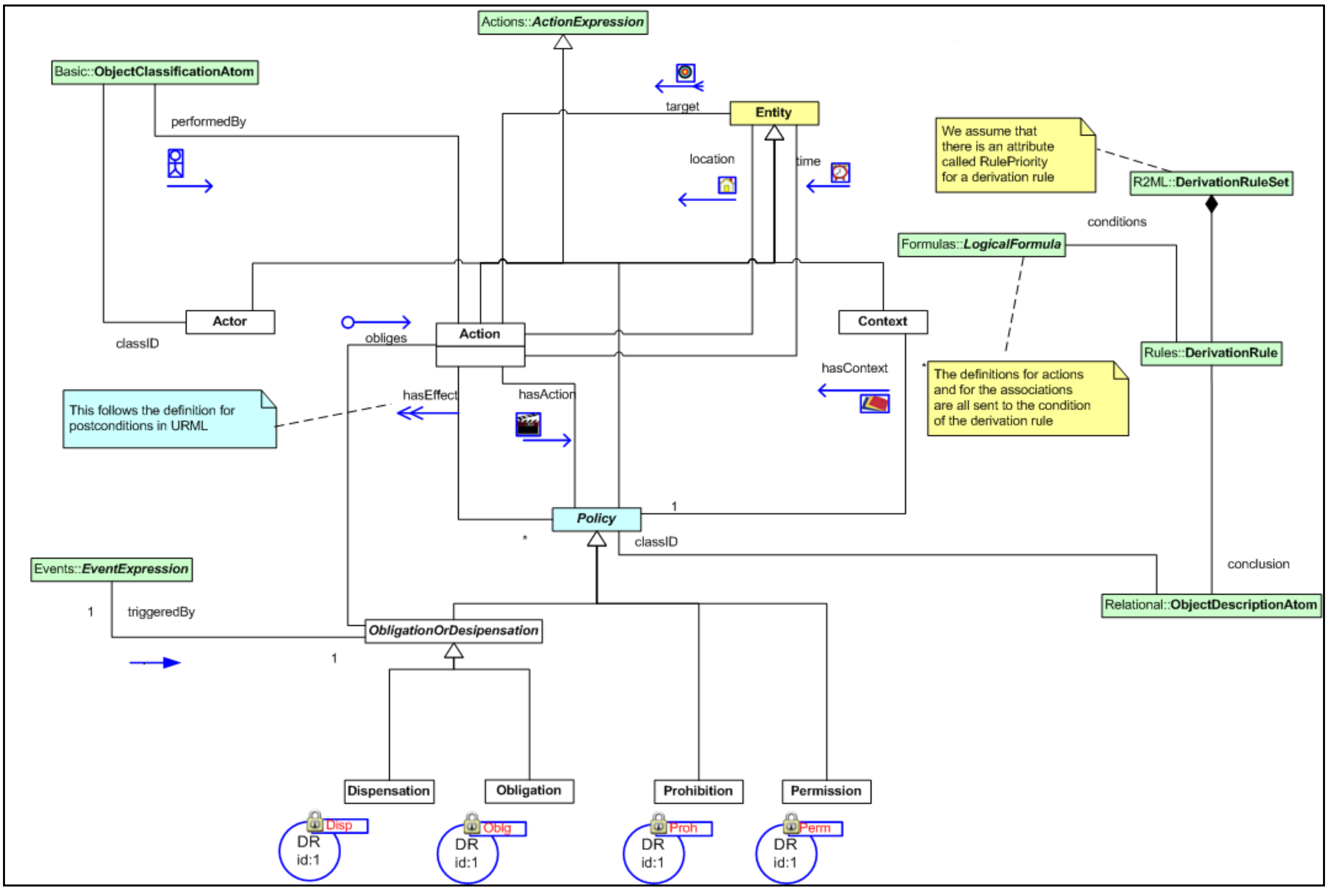

Actions::ActionExpression
Basic::ObjectClassificationAtom
performedBy
target
Entity
location
time
We assume that there is an attribute called RulePriority for a derivation rule
R2ML::DerivationRuleSet
conditions
Formulas::LogicalFormula
Actor
classID
obliges
Action
Context
hasContext
Rules::DerivationRule
The definitions for actions and for the associations are all sent to the condition of the derivation rule
This follows the definition for postconditions in URML
hasEffect
hasAction
Policy
1
*
classID
conclusion
Relational::ObjectDescriptionAtom
Events::EventExpression
1
triggeredBy
1
ObligationOrDesipensation
Dispensation
Obligation
Prohibition
Permission
Disp
DR
id:1
Oblg
DR
id:1
Proh
DR
id:1
Perm
DR
id:1

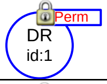
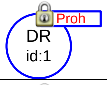
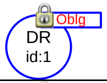
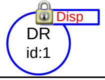
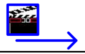
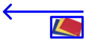
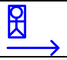
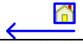
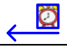
Figure 93. PML metamodel

The condition part of each policy element holds the information about the actions, on which the deontic modal concepts operate, the actors for those actions, and the context of the action. By using R2ML's `ObjectClassificationAtom`, we represent the class to which each object belongs and by using the R2ML's `ReferencePropertyAtom`, we represent association relations in PML. The metamodel has the following associations: *performedBy*, *hasAction*, *hasContext*, *hasEffect*, *obliges*, *triggeredBy*, *location*, *time*, and *target*. The *performedBy* association relates an action as the subject to an actor as the object. The *hasAction* association relates a policy as the subject to an action as the object. Similarly, the *hasContext* association relates the policy element as the subject to the context to which the policy is applied. It can also represent the context in which an action is performed. The *hasEffect* element represents effects of applying a policy, for example, to impose a penalty action. The *triggeredBy* association represents the action upon its occurrence the policy is fired. *Location* and *time* respectively represent the location and the time where the policy occurs. Finally, the *obliges* association is specific to the obligation policy rules and represents the obligatory task that an *actor* ought to perform upon execution of the policy. All these associations from the PML metamodel can be represented by R2ML's `ReferencePropertyAtom`. This preserves the compatibility of PML with R2ML.

2.4.3.3.2. Policy UML Profile

The Policy UML Profile is a graphical concrete syntax of PML. Extending URML, we have developed a graphical concrete syntax for PML (see Table V). To do so, we have defined a UML stereotype for each PML concept.

Table V. PML concrete graphical syntax

Stereotype	UML 2.0 metaclass	Graphical notation
Permission Policy	Class	
Prohibition Policy	Class	
Obligation Policy	Class	
Dispensation Policy	Class	
Action	Class	<<Action>> **ActionClass**
Actor	Class	<<Actor>> **ActorClass**
Context	Class	<<Context>> **ContextClass**
triggeredBy	AssociationEnd	⟶
hasEffect	AssociationEnd	⇇
hasAction	AssociationEnd	
hasContect	AssociationEnd	
obliges	AssociationEnd	○⟶
performedBy	AssociationEnd	
location	AssociationEnd	
time	AssociationEnd	

As an example of the use of General Policy UML Profile, let us consider a policy in which "*Only the doctor from the emergency section is allowed to access the medical test results of a patient and a notification email is required to be sent to the patient*". Figure 94 shows how the Policy UML Profile represents the above policy. The Action class has been used as a stereotype to define the *SendMail* and *AccessElectronicHealthRecords* actions. The *Doctor* is derived from the class *Actor* and is connected to the *AccessElectronicHealthRecords* action via a *performedBy* association. Also the target for the policy has been represented using the class *MedicalTestResults* and the context for the policy has been identified as *ElectronicHealthRecord*. The *EmergencySection* has been represented as an entity connected via a *location* association to the action, thus identifying the place where the policy is applied. As shown in Figure 94, the graphical notation for the PML tries to provide an intuitive way of representing the intentions of the policy designers by adjusting the icons and relations with respect to designers' mental models.

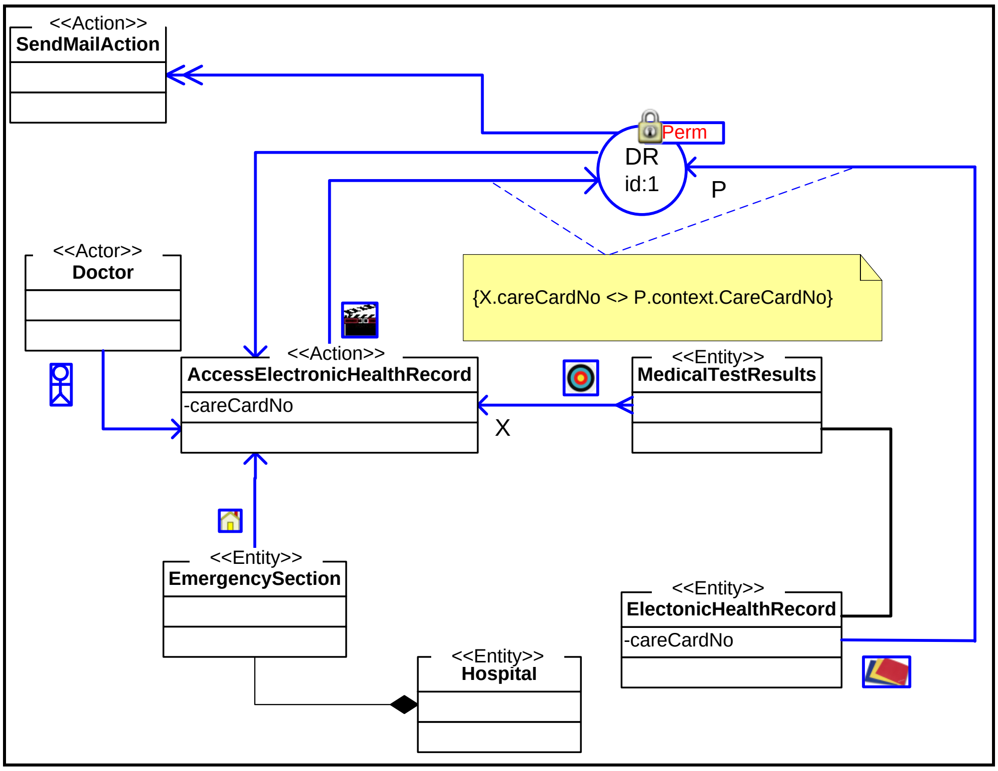


Figure 94. PML model of the patient/doctor policy

2.5. Integration of business rules and business processes

The most notable efforts with respect to the integration concept of business rules and business processes can be associated with the the introduction of a rule modeling method of business processes in 1998, by *Kappel et. al* [54] and in 2000, by *Knolmayer et. al* [59]. The former group of authors proposed using reaction rules in modeling coordination in workflow systems, as reported by [152]. The first time when integration of business rules and processes approaches was identified in information system development is in the work of *Korgstie et. al.* [60]. The authors proposed this integration for capturing temporal information by using the External Rule Language (ERL). ERL is based on first-order temporal logic and it is used to define process logic, so that rules can constraint or describe processes. ERL supports definition of constraint (integrity), derivation and action (reaction) rules. *Korgstie et. al.* also showed how such models of processes could be translated into C source code. *McBrien & Seltveit* [76] extended Korgiste et al's work by proposing a technique for defining rules structure inside a process model. They used process modeling language for defining how activities interact, while business rules (defined in ERL) are used to make precise statements about certain activities. Rules are defined in a form of a reaction rules, and the show how to translate a process to the ERL rules.

Kovacic [64] proposed a metamodel-based approach for linking business constructs, such as process and activities with technical constructs, such as software components. *Knolmayer et al.* [59] created a framework where ECA business rules are used for decomposition of business processes. They showed how business process could be expressed as business rules. In order to allow for the administration of the relations between process and workflow models, Knolmayer et al developed a rule repository, which provides tools for process, workflow and data modeling, as well as import and export capabilities to different workflow modeling systems. However, it is reported that "models that are only composed of reaction rules, have to risk to being over-specified and can be regarded as prescriptive" [40].

Charfi & Mezini [17] proposed a hybrid approach for integration of rules and process by using an aspect-oriented BPEL dialect called AO4BPEL. These authors investigated how different kinds of business rules could be implemented by means of aspect-oriented constructs in AO4BPEL, such as *before*, *after* and *around* advices. The implementation of a business rules is separated from the rest of the process and in the analysis phase, business rules are expressed declaratively as if/then statements. In the implementation phase, each business rule is mapped to business process constructs and to aspects (i.e., "point-cuts", statements that relate an aspect to a specific point in the code). This approach requires a modified BPEL engine to handle such additional aspect-oriented constructs. This approach has been generalized not only to BPEL by *Cibran & Verheecke* [18]. They proposed that business rulse be transformed to aspects, while activities are considered points to place point-cuts. This enable that business process constructs change behaviour in the point-cut places.

Meng et al. [80] developed a dynamic inter-organizational workflow system to integrate e-services. This dynamic workflow model enables the specification of dynamic properties associated with a business process models expressed in Workflow Process Definition Language by adding connectors, events, triggers and rules as modeling constructs. Their system uses a rule server to trigger business processes during the enactment of workflows in order to check business constraints.

Rosenberg & Dustdar [113] proposed a middleware service called Enterprise Service Bus (ESB) for integration of business rules and business processes represented in BPEL. On ESB as a middleware, they attached a BPEL engine, transformation engine, rule interceptor, business rule broker and Web service gateway. The business rule broker is an abstract API for different rule engines, so that they could be used in uniform way with ESB as pluggable interfaces, just by defining a new adapter. The rule interceptor service intercepts all incoming and outgoing messages and automatically applies business rules to them. In this approach, rules are attached just *before* and *after* to the BPEL activities.

Orriëns et al. [97] generate business processes dynamically by composing Web services, if they are constructed and governed by business rules. Business rules are used in the context of service composition to determine how the composition should be structured and scheduled, how the services and their providers should be selected and how run-time service binding should be conducted. They developed a rule-driven mechanism to govern and guide the process of service composition in terms of five broad composition phases including abstract definition, scheduling, construction, execution and evolution to support dynamic business process building. Based on these phases they analyze and classify business rules and determine how they affect service compositions; in this way the composition engine can construct process descriptions.

Similar to *Orriëns et al.*, *Lee et al.* [69] proposed a theoretical foundation where process flows are constructed by means of connecting ECA rules and translated into executable business process, and *Bry et al.* [15] follow the same principle, by suggesting using ECA rules represented in the XChange language for busienss process description. Therefore, a complete business process is represented by using XChange, and every rule is represented as an activity that is checked by invoking certain service. A limitation of this approach is that there is no suggested graphical representation of rules in a process diagram, and in this approach, rules are separated from a process.

Goedertier & Vanthienen [39] use deontic assignments (obligations, permissions and prohibitions) to model business proctols. They have developed an algorithm called PENELOPE that generates process flows for each business partner by applying reasoning over Event calculus. In their notation, decision points are represented as circles, and they group sets of reaction rules. The events in such rules are the inflowing arrows into a decision point. The activities are represented as arrows that flow out of a decision point into the activities. However, PENELOPE only allows expressing business rules about sequence and timing constraints. *Goedertier et al.* [40] further develop their concept from [39] by introducing the EM-BrA^{2}CE framework that addresses control flow, data and resource aspects of business processes. This concept is realized by 16 different types of business rules that need to be enforced during the lifecycle of an activity. They extended SBVR for declarative business process modeling. The EM-BrA^{2}CE framework can undergo multiple state transitions and it allows to construct a suitable execution plan at runtime via different transitions (like OWL-S and WSMO).

Graml et al. [43], propose how business rules can be combined with business processes (BPEL) to run-time change of a running business process. Graml et al. showed how to use derivation rules for controlling flows, data constraints and process rules, to dynamically add tasks or to call subprocesses. In order to support their proposal, they created a central dispatcher process that triggers rule execution and invokes relevant process fragments by using IBM WebSphere Integration Developer.

We should also mention work done in area of distributed rule execution made by *Rosenber & Dustdar* [115] and *Schmidt* [118]. *Rosenber & Dustdar* proposed an approach that uses a rule engine wrapper in order to distribute the control flow over different systems. Execution of business rules forms a process flow. *Schmidt* uses a SOAP messages to execute business rules in distributed environment. He proposes storing business rules in the SOAP header and sending them with messages. This allows distributed execution of the business rules.

Eijndhoven et al. [30], proposed integration of ECA rules into a business process (BPMN-based) by reusing service interaction patterns. Their solution uses ECA rules to model the flow of the process and to execute parts of the process at different variability points. In this way, they implement the same workflow patterns as traditional business process languages. They also proposed methodology consists of three steps: 1. Identify the variable and non-variable (changeable) segments in a process, 2. Identify an appropriate worksflow patterns that model the behavior of each variant in a variation point, 3. Implement workflow patterns using business rules.

In Table VI, we show comparison of abovementioned rules and processes integration approaches. Generally, the research efforts shown in this table can be divided into two major categories: i) fully rule-based; and ii) integration of business rules into process-oriented models (so called hybrid approaches). *In the first group of approaches*, the researchers aim to model business processes fully by using business rules. The representatives of this approach are: Knolmayer [52], McBrien & Seltveit [68] and Bry et al [15]. This approach is usually done with production and reaction rules. JBoss' Drools rules are used for business process modeling, and consequently for regulating service. However, there are a few issues with such an approach: comprehension of the overall process and relations among its constitute parts is tedious given that business rules only focus on small parts of business logic; business process execution is fully driven by reasoning algorithms (e.g., Rete), which might lead to some unexpected behavior hard to determine upfront and might affect the trust of business users in such solutions; there is no effective and unified modeling support of different types of rules; and rules are typically represented in implementation languages, without features to use high-level business process modeling languages.

The second category of (hybrid) approaches recognizes the above problems and proposes methods for integration of business rules and business process modeling languages. Typical representatives of this approach are Eijndhoven et al. [25] and Graml et al. [37]. However, neither of these two solutions proposes a systematic definition of a rule-based business modeling language, which encompasses expressivity of the state of the art languages for both business rule and process modeling.

Table VI. Comparison of rules and processes integration approaches

Approach	**Purpose**	**Reported benefits**	**Process-oriented language**	**Rule language**	**Integration of languages**	**Types of rules**			
						IR	DR	PR	RR
McBrien & Seltveit [76]	Description of a process by means of business rules.	Process representation of a rule model, formal and graphical representation of rules, executability of a process model by code generation.	PID	ERL	Graphical syntax	+	+	-	+
Knolmayer [59]	Modeling business processes by means of ECA business rules.	Integration platform for different process modeling techniques, partial modeling of different process actions by means of business rules.	Extended ECAA notation	ECA notation	Graphical syntax	-	-	-	+
Charfi & Mezini [17]	Web services composition by use of AOP as an implementation technology to integrate rules into business processes.	Dynamic adaptation of the web service composition, integration of rules into a service composition .	BPEL	AO4BPEL	Hard-coded via aspect oriented programming	+	+	-	+
Meng et al. [80]	Integration of e-services in inter-organizational workflows.	Dynamic workflow model for modeling business processes, dynamic service binding, the technique for run-time modifications of the run-time workflow structures to alter the course of executing workflow instances.	WPDL	Hard-coded (CBRSL)	Graphical syntax and Metamodel-based	-	-	+	-
Rosenberg & Dustdar [113]	Integration of rules in a BPEL.	Integration of business rules in process-oriented Web service composition, use of standard Java Rule API for integration of different rule engines.	BPEL	Java, RuleML	Web service based and hard-coded	+	+	-	+
Orriëns et al [97]	Business rule driven service composition framework.	Generation of business process dynamically by composing Web services constructed by business rules. The rules contain facts on the process elements and their required flows, based on which the composition engine is able to construct the flows and the elements into a process description that can be then executed..	AGFIL-BM	RuleML	Graphical and textual syntax based	+	+	-	+

Bry et al [15]	Use of ECA rules for describing business processes in an executable manner.	Realization of business process control flow by means of ECA business rules.	XChange	XChange	Hard-coded (procedural)	-	-	-	+
Goedertier, Haesen, Vanthienen [39] [40]	Using business rules to represent policies and regulations in business process models.	Generation of process flows for each business partner by applying reasoning over Event calculus.	EM-BRACE	PENELOPE for deontic assignments and Hard-coded for other types of rules	Graphical syntax and Metamodel-based	+	+	+	+
Graml et al. [43]	Integratio of business rules with business processes to run-time change of a running business process.	Set of business rule-enabled business process modeling patterns that overcome adaptability limitations for process decisions, constraints and subprocesses. Extraction of business logic that is contained in business process models into business rules.	BPEL	Java	Graphical syntax and hard-coded	+	+	+	-
Eijndhoven et al. [30]	Uses of business rules and workflow patterns to model the variable parts of process flow, by facilitating dynamic pattern composition.	Solution to increased flexibility of service oriented business processes by using ECA rules to execute parts of the process at variability points. ECA rules are used to model the flow in a business process.	BPMN	iLOG JRules	Graphical syntax (transformations)	-	-	-	+

3. Rule-enhanced Business Process Modeling Language and Methodology

Service-oriented architectures (SOA) offer flexible integration of software systems and various collaborating parties. Service development mainly follows proven principles of business process modeling. Based on process-oriented modeling languages (e.g., Business Process Modeling–BPMN notation and UML Activity Diagrams), service developers start from the high-level business process models. Such models typically cover aspects such as flow of control, data, and activities. A significant progress in this area has been made in several critical aspects. In spite of the promising results, there are several research challenges to be addressed. *First*, process-oriented models are typically not well-connected with models of software structure or business vocabularies. This reduces the extent to which service definition can automatically be generated from the process models (e.g., definitions of message types). *Second*, constant changes of business logic require effective mechanisms for updating of process models. Currently, there is limited support for decoupling parts of business logic (e.g., constraints and process decisions) from the full process models. Consequently, this limits more dynamic updates of business logic and their propagation to the executable software systems. *Third*, the complete business process models are typically hard to be understood by business experts. Capturing smaller chunks of business logic and their consequent integration into bigger process models is a more effective and reliable approach. *Finally*, current process-oriented models do not provide a complete support for modeling of exchange of business logic between different collaborating parties. For example, in some situations, a service requestor wants to subscribe for a certain services, which will be only invoked under certain conditions (e.g., currency exchange rate change). Service providers need to obtain such rules from the service requestor, check if such rules are valid w.r.t. their service policies, execute such rules on their (service) side, and allow for dynamic rule changes by service requestors.

In order to answer to these challenges, we propose a language called Rule-based BPMN (rBPMN) that integrates support for business rules (R2ML) in business process modeling (BPMN). Previous work on this topic provided some promising research results (see Section 2.5). Probably, the most comprehensive approach is [42][43] that identified a set of patterns for integration of business rules into business process models. However, that work did not provide a definition of a rule-based process-oriented modeling language. Consequently, it was not possible to provide a comprehensive solution that addresses the four previously mentioned challenges.

Additionally, business rule approach can bring more benefits to business processes, such as:

- Business rules extract business logic in form of conditions form business processes, which enable flexibility to make changes in a business process.
- As business rules can be stored in one place (repository), they can be reused in multiple places in a business process.
- Such rules are more understandable by business people, which could be less error-prone approach in defining business processes.
- The business rules enabled declarative approach, and that is one is focused on what *should* be solved and not *how*.

In this section, we define a rule-based business process modeling language–rBPMN. rBPMN is a result of integration of the BPMN and R2ML languages. BPMN has been selected due to its broad user adoption, comprehensiveness in covering business process concepts, and rich experience in its use. The selection of the rule language was driven by: research objective specified in the previous section; need to make use of a proven and rich rule modeling language; previous experience in integrating with software modeling languages; and objective to follow proven principles and standards for engineering software modeling language (i.e., model-driven engineering). Thus, R2ML has been selected, as other relevant languages are used only either for rule interchange (e.g., RuleML and RIF) or for representing a specific type of rules (e.g., SBVR and PRR). R2ML is defined by using model-driven engineering principles and its relations with UML have been deeply investigates. Briefly, rBPMN is defined through integration of R2ML and BPMN at the level of their metamodels (so-called model weaving).

We have shown that [105] the combination of BPMN and R2ML gives the highest ontological completness among different languages we analysed: BPMN 2.0, PRR, R2ML, SWRL and rBPMN. This analyse is done by using well-known Bunge-Wand-Weber (BWW) representation model [144]. Ontological completeness means that a user of a given language is able to represent a relevant real world scenario when modeling in the given modeling language. We analysed multiple possibilites for combining process and rule modeling languages in order to achieve higher ontological completness. More details about representational analysis of business process and rule languages can be found in [105].

In the following subsections, we first explain extensions to its graphical syntax, and then explain integration of metamodels of BPMN and R2ML.

3.1. rBPMN graphical concrete syntax

In order to support usage of R2ML rules in BPMN, we decided to include rules in two different ways, as events and as gateways. Primarily, our approach is based on representing rules as gateways in rBPMN, because rule decisions can be represented as forks in business processes, i.e., outgoing sequence flows from a gateway. As our intetion is to represent rules as first-class citizens in rBPMN, we introduced a new gateway symbol, called *rule gateway* (a gateway with R symbol inside). As R2ML support four types of rules: reaction rules, production rules, derivation rules and integrity rules, we wanted to introduce rules uniquelly in rBPMN, so rule gateway can be connected to any of these rule types. The rule gateway can be connected to one rule, two or more rules, or to a ruleset.

For representing rules and their conditions/actions, we use graphical syntax from URML language (see section 2.4.3.2). In table VII we show how URML graphical elements are mapped to the R2ML metamodel elements.

The complete rBPMN synax for rules can be found in Appendix D.

Table VIII. Mappings between URML and R2ML elements

URML Description	URML Metamodel	R2ML Metamodel
Derivation Rule: a circle with a label “DR” and a rule identifier.	DerivationRule	DerivationRule
Production Rule: a circle with a label “PR” and a rule identifier.	ProductionRule	ProductionRule
Reaction Rule: a circle with a label “RR” and a rule identifier.	ReactionRule	ReactionRule
Association Condition: an arrow from an association to a rule.	BinaryAssociationAtom	ReferencePropertyAtom
Classification Condition: an arrow from a class to a rule.	ClassificationAtom	ObjectClassificationAtom
Property Condition: an arrow from a property to a rule.	RoleTypeAtom	PropertyAtom
Post Condition: a double-head arrow from a class to a rule.	ClassificationAtom	ObjectClassificationAtom
Association Conclusion: an arrow from a rule to an association.	BinaryAssociationAtom	ReferencePropertyAtom
Classification Conclusion: an arrow from a rule to a class.	ClassificationAtom	ObjectClassificationAtom

Property Conclusion: an arrow from a rule to a property.	RoleTypeAtom	PropertyAtom
Attribution Conclusion: an arrow from a rule to an attribute.	AttributionAtom	AttributionAtom
Assert Action: a double-head arrow from a rule to a class, with an “A” near the arrow head.	AssertActionExpr	AssertActionExpression
Retract Action: a double-head arrow from a rule to a class, with an “R” near the arrow head.	RetractActionExpr	RetractActionExpression
Update Action: a double-head arrow from a rule to a class, with an “U” near the arrow head.	UpdateActionExpr	UpdateActionExpr
Invoke Action: a double-head arrow from a rule to a class, with an “I” near the arrow head.	InvokeActionExpr	InvokeActionExpression

We should also mention that, in the standard BPMN [89], there is Conditional Event Definition (rule event), which can be used to attach some expression defined in a rule language, but this event type models only the behavior of production rules. In addition, it is possible to use an expression attached to the outgoing conditional sequence flow when the source is gateway, however, in the standard BPMN, there is no concrete proposal for a rule language that could handle such expressions.

In addition, we connected a production rules to the BPMN Conditional (rule) event, because in standard BPMN this type of event is not associated with any formal rule language. We need to have a formal language because we want our process to be executable. Reason for using production rules is that this type of event is triggered when its condition evaluates to true, so production rules can support this kind of behavior.

In subsequent sections, we will how we integrated rule gateway and rule event with different rule types, as well as definition of vocabulary that we use in rules.

3.1.1. Vocabulary in rBPMN

As each rule needs to have facts (data) in order to infere and to return results, we need to define a basic vocabulary that will be used by a rule. An R2ML vocabulary is defined in the form of UML-class diagrams, by using basic vocabulary concepts introduced in Section 2.4.3.1.5. Similariy, we use R2ML vocabularies in rBPMN to represent messages and data.

In Figure 95, we show an example of a R2ML vocabulary from the EU-Rent Vocabulary Business Context [32] defined in rBPMN. EU-Rent is a car rental company owned by EU-Corporation. EU-Rent rents cars to its customers. Customers are individuals. In this figure, we can see how we graphically represented classes (such as Rental, Person), class attributes (e.g., forename, surname, and address attributes with the String datatype in class Person), inheritance (classes BarredDriver and QualifiedDriver are subclasses of the class Person), and reference properties (e.g., badExperience reference of the class Person). We should also mention that R2ML vocabulary enables defining datatypes, operations, variables (data and object) and constants, too.

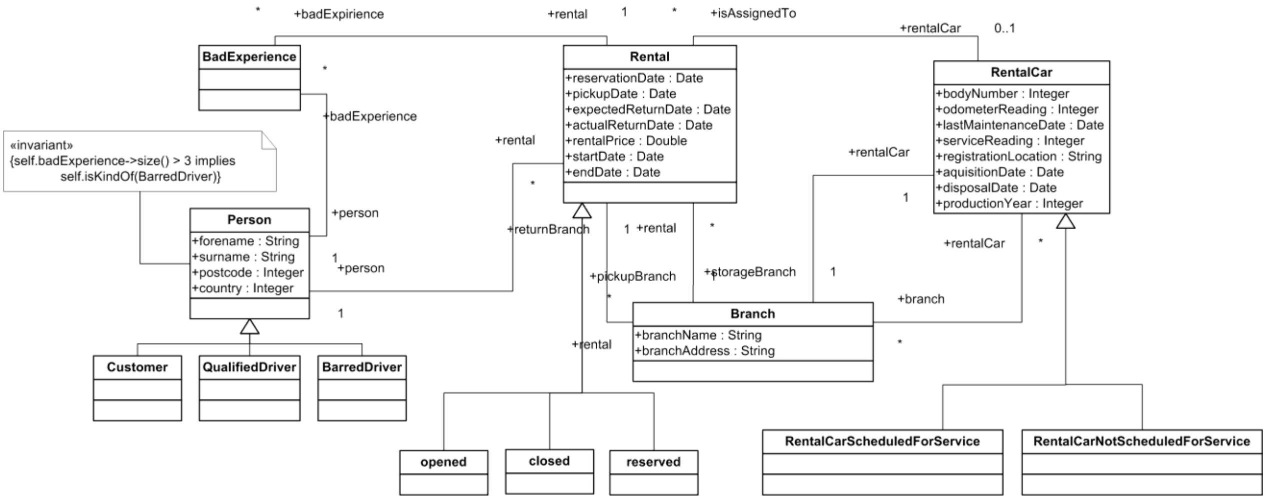


Figure 95. An example of R2ML vocabulary excerpt defined in rBPMN [32]

Each of the classes shown in Figure 95 can be separately used in defining rule conditions.

3.1.2. Integration of R2ML rules and rBPMN rule gateways

A rule gateway can be associated to one or more the R2ML rules where rules can be of different rule types. However, with respect to the rule type, the rule gateway meaning remains the same, but its context is different. In order to see how the rule gateway can be connected to different rule types, we will present that integration in this section for each R2ML rule type.

3.1.2.1. Integration of integrity rules and the rule gateways

First, we will show how R2ML *integrity rules* are associated with a rule gateway. In URML integrity rules are represented as OCL invariants on a data model. Also, our previous research demonstrated there is a bi-directional transformation for mapping between R2ML integrity rules and OCL [94]. Thus, we connect a rule gateway to one or more OCL invariants. Also, one OCL invariant can be referenced by multiple rule gateways. In Figure 96, we show how we connect an integrity rule in form of an OCL invariant with a rule gateway (the invariant is attached to the Person class in Figure 95). This invariant states that *a barred driver is a person known to EU-Rent as a driver who has at least 3 bad experiences*. In this case, the rule gateway use attached integrity rule to infer whether the invariant evaluates to true or to false. The attached invariant, use classes from a predefined common data model (vocabulary) in its definition, so the incoming data in the rule gateway' incoming data flow must have the same variable type in order for rule to conclude (the mappings between process data and rule data is defined in the rBPMN editor). Based on the rule return value (Boolean decision), the corresponding outgoing sequence flow from a rule gateway is chosen. The crossed outgoing sequence flow from a rule gateway is chosen in the case of a negative decision (the False task in Figure 96), and the other sequence flow, which is not crossed, is chosen in the case of a positive decision (the True task in Figure 96). The implementation logic for a rule gateway must be supported in the process execution engine. We can have one or more integrity rules connected to the rule gateway, so we can one or more outgoing sequence flows from a rule gateway. If any of invariants evaluates to false, then the condition is false and thus the False task is executed. Otherwise, if all of them are true, then the True task is executed. This is the case because integrity rules usually represents constraints in a process, which when not satisfied a process flow cannot continue, and it forks in that point (usually handled by some exception handler). The task for an exception handler is to determine which invariant is violated and consequently make corrective actions. If invariants are transformed into SWRL or OWL, we can use a reasoner to provide an explanation why some error has happened or some incosistency has occured. Thus, it is easy for the exception handler to know how different invariant violations to handle.

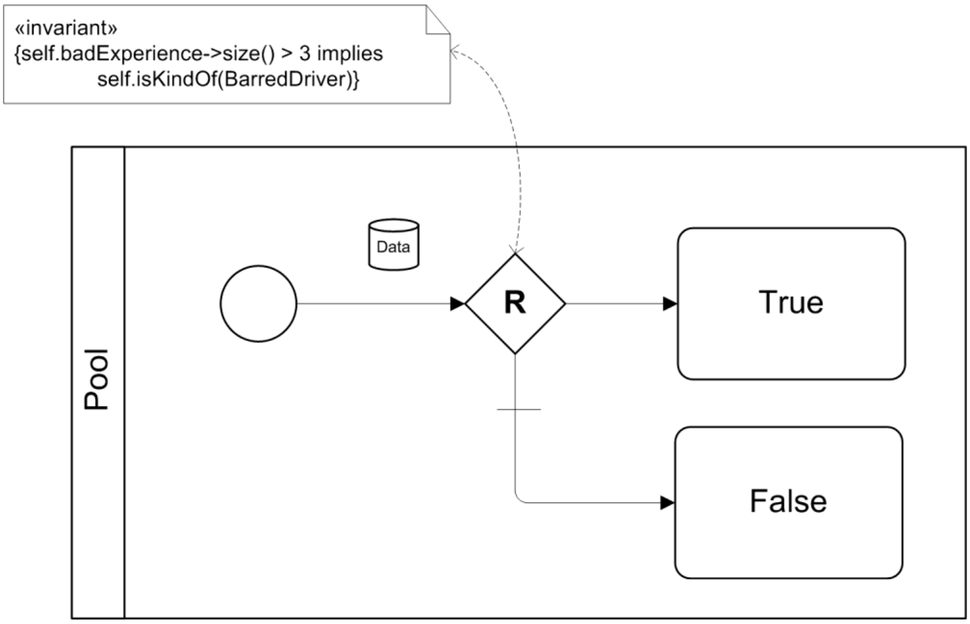

«invariant»
{self.badExperience->size() > 3 implies
self.isKindOf(BarredDriver)}
Data
R
True
False
Pool

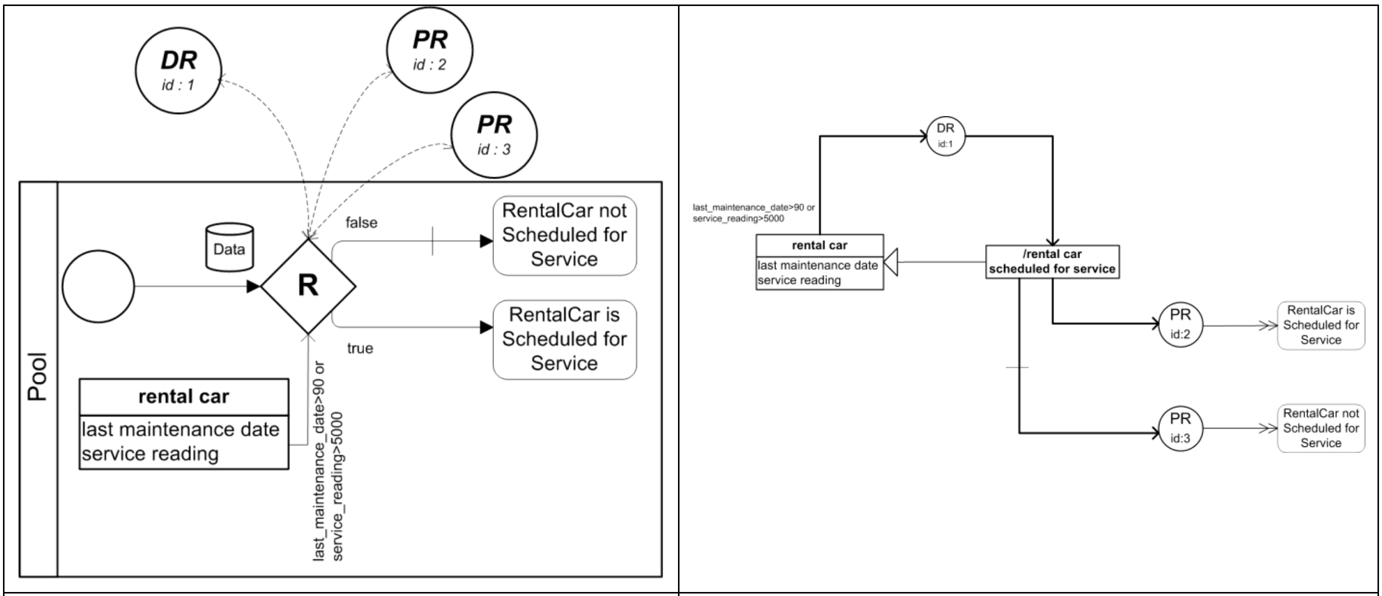
Figure 96. Integration between integrity rules and the rule gateway (R)

3.1.2.2. Integration of derivation rules and the rule gateway

Integration of *derivation rules* and rule gateways is different from the integration of integrity rules and rule gateways. The main difference is that in case of derivation rules, *besides* derivation rules we must have also production or reaction rules connected to the rule gateway. We use a derivation rule to infere new knowlegde and if the rule condition is satisfied the rule will produce a new fact. Then, we use associated production (or reaction) rule that will be triggered to make a decision. In addition, derivation rules could be used without production or reaction rules, when there is no connection to the rule gateway, e.g., derivation rule can be attached to an activity to infere some fact that can be used later in the process.

We use two rule in the case of Boolean decision, and two or more rules in the case of multiple choice. Two rules are needed because of a rule nature [143], where condition on one rule must be negated condition of another rule.

In Figure 97, we can see how a derivation rule and to production rules connected to a rule gateway in the case of *Boolean decision*. The condition attached to the rule gateway is a condition (*if rental car last maintenance date >= 3 months or if rental car service reading >= 5000 km then rental car scheduled for service*) of the derivation rule, If the condition on the derivation rule evaluates to true, the “true” the fact ”rental car scheduled for service” is produced. In this case, the first production rule (id:2) is chosen to execute the “RentalCar is Scheduled for Service” task. If the condition evaluates to false, then the second production rule is invoked and the “RentalCar not Scheduled for Service” task is invoked. Therefore, each rule gateway outgoing sequence flow is connected to a production rule. The data from a business process is passed to the rules’ condition part and it is defined as part of the rBPMN (R2ML) vocabulary.



a) rBPMN model for rental car scheduled for service decision

b) Corresponding rules

Figure 97. Integration between derivation rules and the rule gateway (R) – Boolean decision

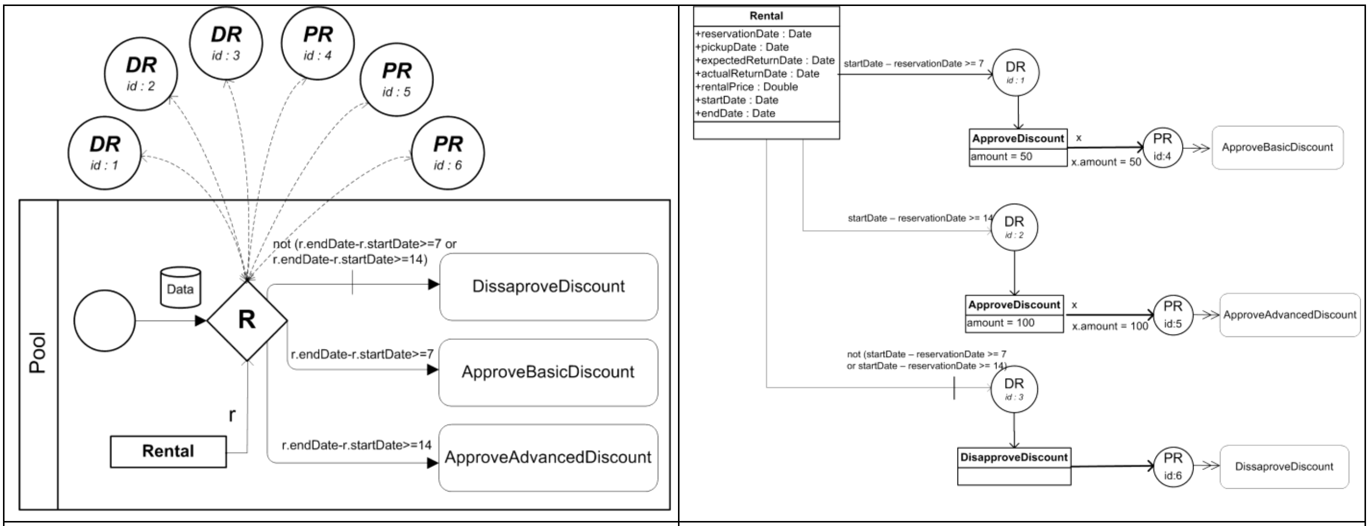
When we have two or more outgoing sequence flows, i.e., in the case of *multiple choice decision*, we can have two variants. In the first case, each outgoing sequence flow from a rule gateway can be directly mapped to a derivation and production rule pair, and in the second case, we can have multiple outgoing sequence flows attached to a production rule, and one (crossed) outgoing sequence flow that acts as ELSE branch.

If we have a rule gateway *RGj* to which we can assign *n* production or reaction rules: *LE = {LE1, LE2, ..., LEn}*, where for each production/reaction rule *LEj (j=1..n)*, which is triggered on a fact that is derived through its associated derivation rule, exists exactly one outgoing sequence flow from a rule gateway. Each outgoing sequence flow (*OFj*) from a rule gateway is choosen if a logical expression *LEi* $1 \le i \le n$ is evaluated to true. However, for each logical expression, we can also have a false result. In that case, we need to have exactly one else for a rule gateway, whose logical expression is *not (LE1* ∨ *LE2* ∨*...*∨ *LEn)*. So, each logical expression on a outgoing sequence flow from a rule gateway is mapped to a R2ML rule condition: *R = {R1, R2, ..., Rn}*. This means that each rule have assigned one logical expression *LEj (j = 1..n)* for its condition.

The multiple choice decision by using three pairs of derivation-production rules attached to the rule gateway represented in rBPMN shown in Figure 98 for a following set of rules:

- *Give discount 50 points for money rental if rental period >= 7 days* (DR - id:1);
- *Give discount 100 points for money rental if rental period >= 14 days* (DR - id:2);
- *Give no discount if rental period is less than 7 days* (DR - id:3).

Each outgoing sequence flow condition from the rule gateway represents condition on one of the production rules attached to it (see Figure 98b). These rules are also triggered on a fact that is derived through its associated derivation rule. Based on the rule's evaluated condition, the appropriate outgoing sequence flow from the rule gateway is chosen.



a) rBPMN model for rental car scheduled for service decision

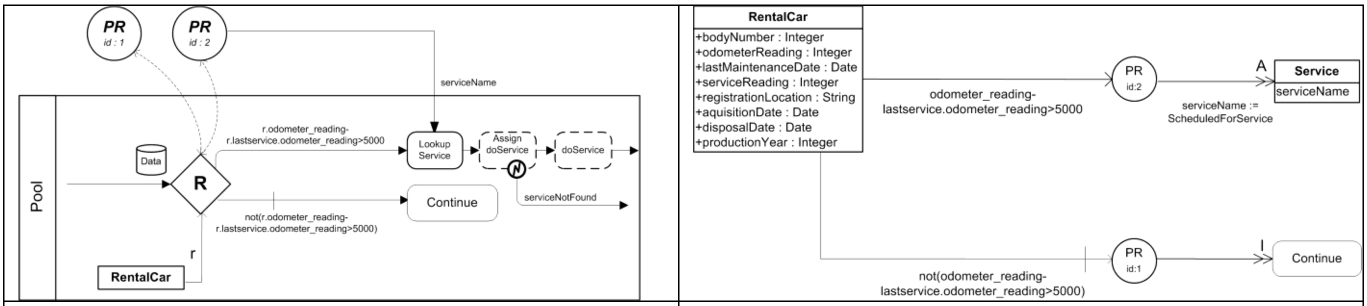
b) Corresponding derivation rules

Figure 98. Integration between derivation rules and the rule gateway (R) – Multiple choice decision

3.1.2.3. Integration of production rules and the rule gateway

In the case when we need to call some action to be executed directly from a rule or to define a postcondition, we can use *production rules* connected to a rule gateway. The integration between production rules with a gateway is the same as between derivation rules and a rule gateway, already described in section 3.1.2.2. However, in this case we can use rules produced action part to invoke, update, assert or retract data in the working memory.

In Figure 99, we show an example of a rBPMN process for a rule that states: *if customer returns a car and the car has more than 5000km from the last service then send the car to the service*. The rule gateway in Figure 99a has a RentalCar for its rule condition from the vocabulary, and two outgoing sequence flows. On the first outgoing sequence flow, we define the rule condition, while on the second (crossed) outgoing sequence flow, we have the same rule but just negated. In Figure 99b, we have two corresponding production rules for the two outgoing sequence flows from the rule gateway (shown in Figure 99a). If the rule (id:1) condition is not satisfied, then the *Continue* activity is invoked, and the crossed outgoing sequence flow is. Otherwise, if the rule condition is satisfied, the rule (id:2) the production rule (id:2) is returning the *Service* message with the service name, added by the rule *append* action, and the *Service* message is used by the *LookupService* activity to find a concrete service with the *serviceName* name and to enable it for the *Assign doService* task to invoke it instead of the *doService* (dummy) task in the process. As shown in this example, this enable for dynamically chosing the invoked activity (service) by using production rules attached to the rule gateway.



a) rBPMN model for rental car scheduled for service decision

b) Corresponding production rules

Figure 99. Integration between production rules and the rule gateway (R)

3.1.2.4. Integration of reaction rules and the rule gateway

Reaction rules are similar to production rules, and they can be used in the same way with the rule gateway, as it can also have a postcondition and it can invoke, update, assert or retract data in the working memory. However, reaction rules have some advantages over production rules. They can be invoked by an explicit event, and they can be triggered or trigger *atomic* or *composite* events (such as sequence of events, parallel events, choice events and events such as, when event A occurs and event B does not).

The reaction rule expression when event A occurs the B event does not (the *AndNotEventExpression* from the R2ML metamodel – see section 2.4.3.1.5.4) can directly be mapped to the output of the rule gateway that have exactly two outgoing sequence flows, where one of these sequence flows is negated (as shown in Figure 97 for derivation rules).

In Figure 100, we show two corresponding reaction rules for the rule gateway given in Figure 97a. Both reaction rules have the same conditions as derivation rules in section 3.1.2.2. However, in this example both reaction rules have an incoming request message. The first reaction rule (id:1) is triggered on the *CheckIsRentalCarScheduledForService* request (event), and if the condition is satisfied, the rule fires *rental car scheduled for service* event (by using double-headed arrow), and at the same time the rule ensures that the *rental car not scheduled for service* event is not fired (by using crossed double-headed arrow). The second reaction rule (id:2) in Figure 100 is triggered on the same request (event), and if the negated condition is satisfied, the rule fires the *rental car not scheduled for service* event (by using double-headed arrow), and at the same time, the rule ensures that *rental car scheduled for service* event is not fired (by using crossed double-headed arrow).

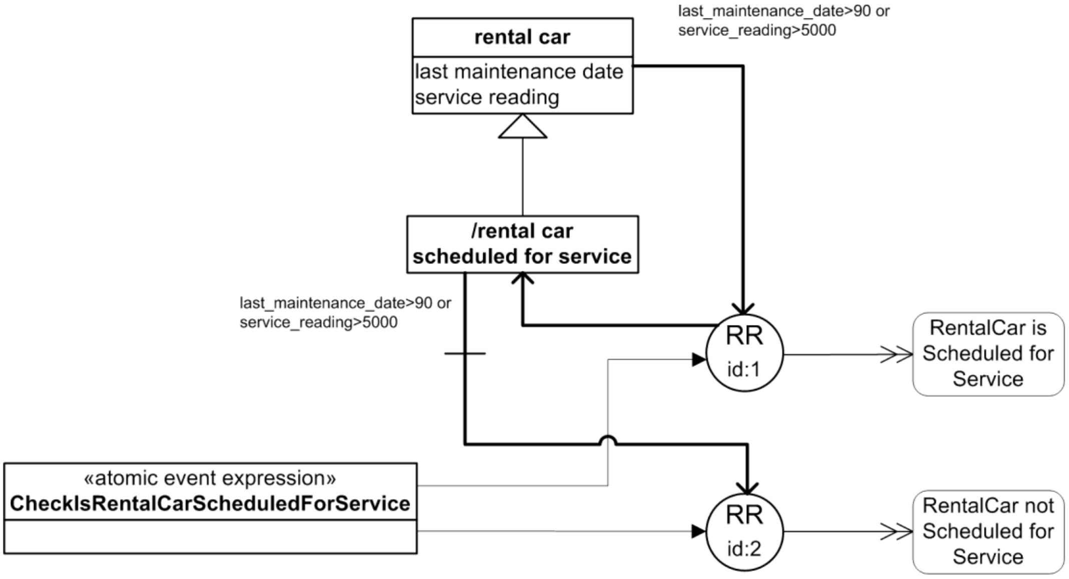
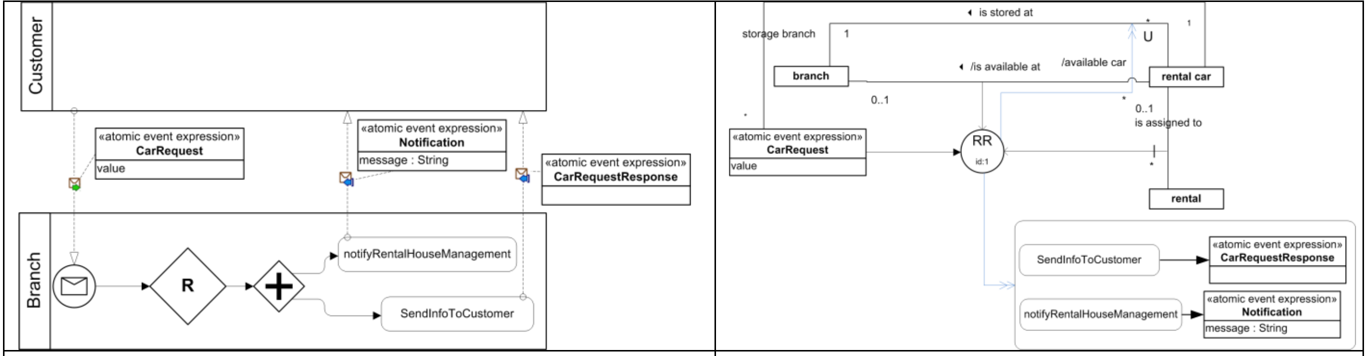

last_maintenance_date>90 or
service_reading>5000
rental car
last maintenance date
service reading
/rental car
scheduled for service
last_maintenance_date>90 or
service_reading>5000
RR
id:1
RentalCar is Scheduled for Service
«atomic event expression»
CheckIsRentalCarScheduledForService
RR
id:2
RentalCar not Scheduled for Service

Figure 100. Reaction rules for the rule gateway shown in Figure 101a

In the case of parallel events, we use reaction rule to trigger them simultanously. We will show how we can use the rule gateway with reaction rules in the case of the following reaction rule: on *a customer car request, if the car is available then send approval and notify rental house management at branch*. In Figure 101a, we show this rule in the form of a rBPMN process. After the start message event (annotated by the *CarRequest* message) has been sent from the *Customer* pool to the *Branch* pool, the rule gateway is enabled to invoke the attached reaction rule (shown in Figure 101b). If the rules' condition (*rental car is available at branch*) evaluates to true, three actions are produced, the rental car is stored at the branch (in data model) and the *CarRequestResponse* and *Notification* messages are generated to be send to the *Customer* pool. In Figure 101a, these the later two actions are done by using the *notifyRentalHouseManagement* and *SendInfoToCustomer* tasks, after the parallel gateway, respectively.

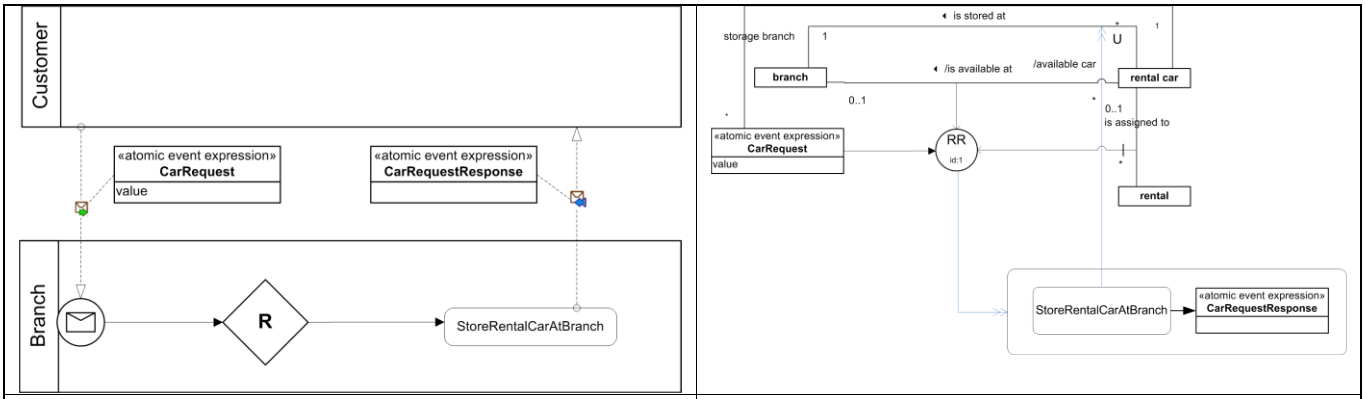


a) rBPMN model for rental car request decision

b) Corresponding reaction rule

Figure 101. Integration between reaction rules and the rule gateway (R) in the case of a parallel events

In the case of a *sequence* composite event, we have multiple events that happen in an ordered sequence. We will use the same example as for the parallel events, but now composed in a sequence. We want *first* to mark the requested car as stored at branch, and then to send approval to the customer. In Figure 102a, we show the same rBPMN process as in Figure 101a, but in this case we introduce a subprocess with two tasks. A new task *StoreRentalCarAtBranch* which is used to store car at branch, and then we send the *CarRequestResponse* message to the *Customer*. In Figure 102b, we have a corresponding reaction rule which is attached to the rule gateway in Figure 102a. In this rule diagram, the reaction rule's action part now triggers a subprocess with the *StoreRentalCarAtBranch* activity, which is used to store the requested car at the branch and returns the *CarRequestResponse* message to the *Customer*.

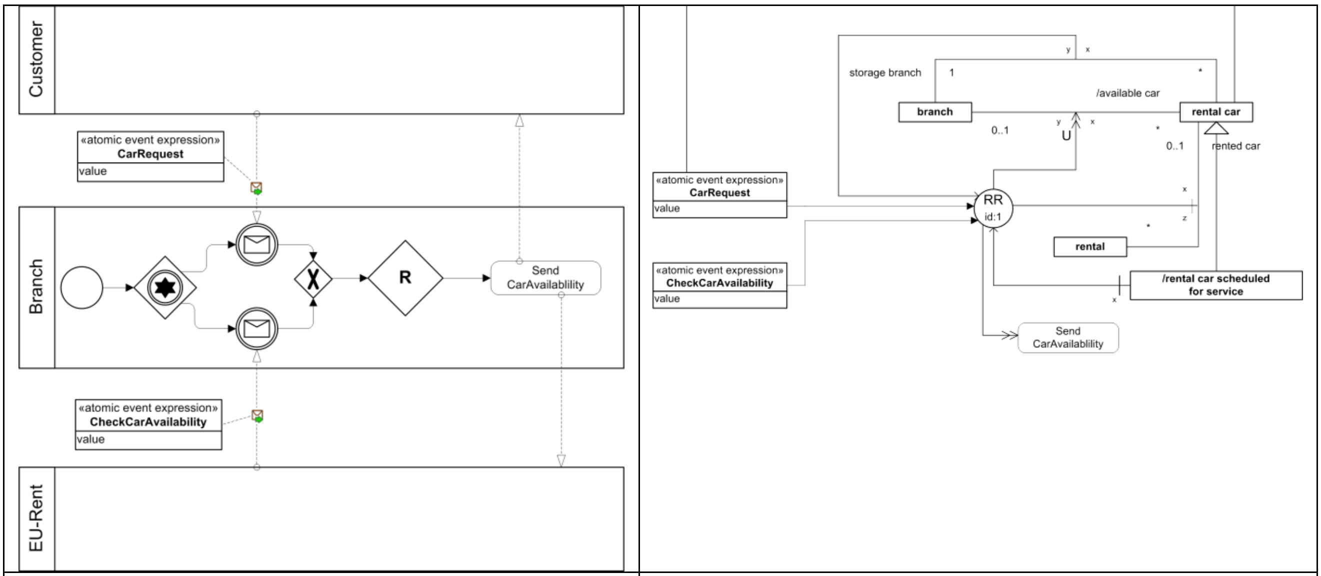


a) rBPMN model for rental car request decision | b) Corresponding reaction rule

Figure 102. Integration between reaction rules and the rule gateway (R) in the case of a sequence events

The usage of the *choice* composite event of R2ML is shown on the following rule example: *if a car is stored at a branch, it is not assigned for rent and it is not scheduled for service, thus mark car as available at the Branch*. A check car availability task can be triggered by a Customer in the case of car a request, but also by the car owner (EU-rent company), for example, in order to transfer the car to a Branch without free cars. This means that we can get two different requests (triggers), but only one event is needed to happen in order to fire a reaction rule. The choice composite event represents triggering events of a R2ML reaction rule.

In Figure 103a, we show an rBPMN process model where *Customer* can send the *CarRequest*, but also *EU-Rent* can send the *CheckCarAvailability* message to the *Branch*. We used an exclusive gateway before the rule gateway in order to choose one or both events. On either of the two events, the reaction rule attached to the rule gateway, shown in Figure 103b is triggered. If the car is available, it is marked as available, and the *Send CarAvailability* task is invoked to return the message to the *Customer* or the *EU-Rent* (depending on who sent the message, by using an implicit correlation). Based on the information returned from the Branch, the *Customer* or *EU-Rent* can decide what further to do with a car.



a) rBPMN model for a car availability check | b) Corresponding reaction rule

Figure 103. Integration between reaction rules and the rule gateway (R) in the case of a choice composite event

3.1.2.5. Integration of PML policies in rBPMN language

In order to integrate policies into rBPMN, we use the Policy Modeling Language (PML) [56], as it is built on R2ML. PML abstracts common policy concepts from several policy languages and by grounding it on the sound theoretical foundation of deontic logic, while keeping it away from the paradoxes that deontic logic introduces [82]. The language can be used in the software analysis and design phases together with other well-established languages (e.g., UML) and can be deployed (implementation and integration phases) and transformed to different policy languages and used with different technologies (e.g., for business vocabularies and rules, components, and processes).

As PML policies are represented as a R2ML Derivation rules and as PML policy can be placed in different places in a business process, we decided to support PML policies by using BPMN associations from three BPMN elements, regarding translation of the PML policies into SOA (WS-Policy language [139]). In Figure 104, we show how PML policies can be attached to different rBPMN elements regarding translation to WS-based elements, by means of (derivation) rule annotations. Namely, we have a support for attaching PML policies to following BPMN elements, as WS-Policy language define policies on WSDL services, portType and message:

a) *Pool* – for defining PML policies on WS Services and Endpoints;

b) *MessageFlow* – for PML defining policies on WS Messages, and

c) *Task* – for defining PML policies on WS Operations.

In Figure 104, we have attached a policy to the Pool first, and this means that this policy apply to all services (activities) offered by this Pool. The second policy is attached to the message flow, and this introduces a message-level security for protecting messages. In the third case, the policy is associated with the Activity. This means that the policy the Activity can be invoked by the requestor only if conditions from the policy are satisfied.

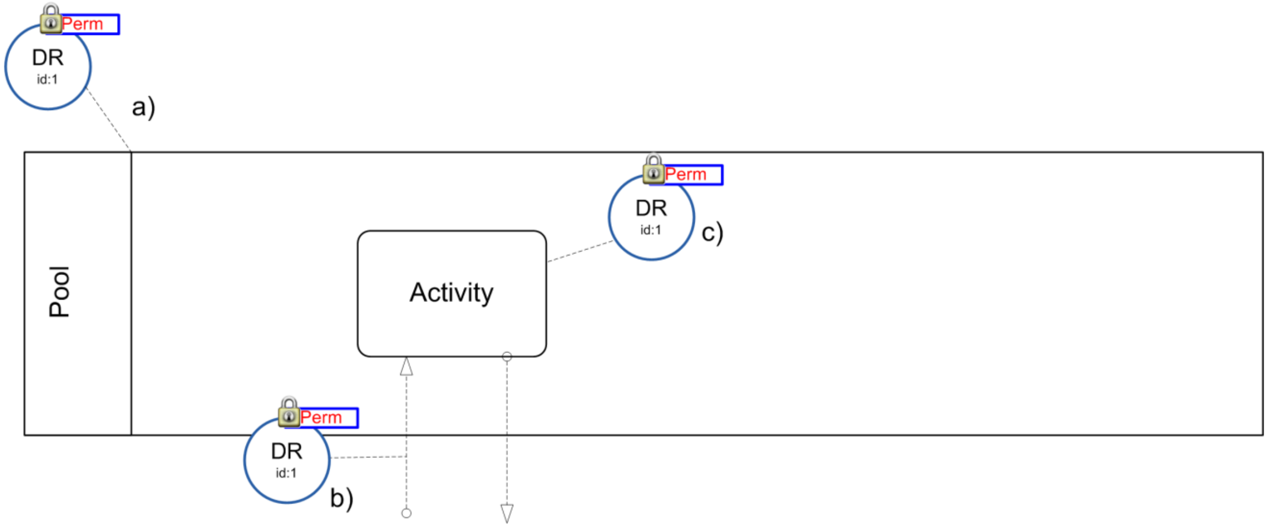


Figure 104. GPML policies in rBPMN language

We use policies in order to convey conditions on an interaction between two partners in an interaction. A provider of a service usually exposes a policy to give conditions under which it provides that servis, and the requester of the service uses this policy in order to decide whether to use the service.

We use WS-Policy here as it polices are attached to WSDL elements, and on the implementation level rBPMN elements are mapped to the BPEL/WSDL elements, as shown in BPMN2 specification [89]. As we have already developed transformations between R2ML and WSDL [111], it is straightforward to add WS-Policy concepts in WSDL Web service generation. Therefore, we can generate complete BPEL processes with WSDL (WS-Policy) descriptions from a rBPMN process.

PML policies, which are actually specialized R2ML Derivation rules, can also be connected to the rule gateways, as shown in Figure 105.

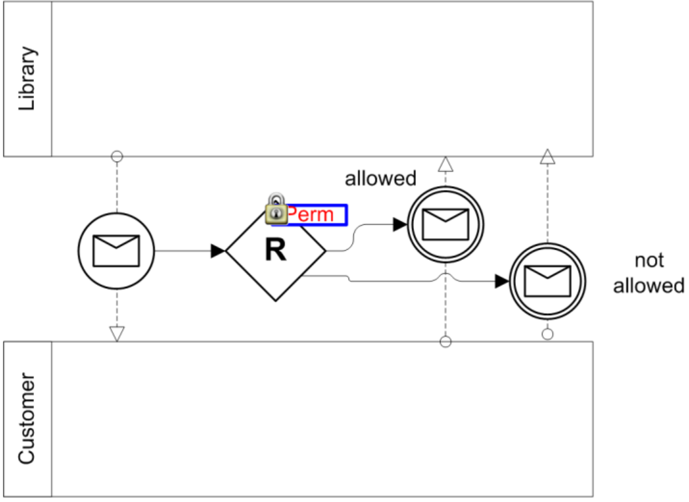


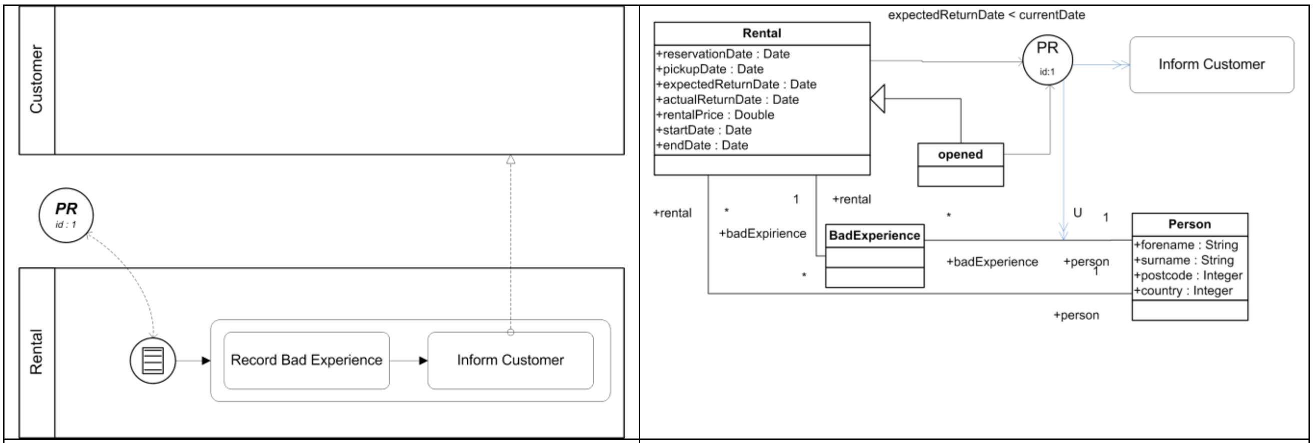
Figure 105. GPML policy in interaction model diagram

3.1.3. Extension of Conditional Event Definition (rule event)

The standard BPMN has a Conditional (rule) event [89], which is triggered when its assigned Boolean condition evaluates to true. It can be used as a start or intermediate event in a process. Additionally, this type of event can be attached to the boundary of an activity, and it can interrupt an execution of the activity. The Conditional event is a type of *Expression* in BPMN. However, Expression is used in the BPMN specification [89] to specify an expression in natural-language. Nevertheless, these expressions are not executable, so we decided to extend them with rules in order to be executable, and to define more formally its behavior.

Therefore, we connected R2ML production rules to the BPMN Condition events in a rBPMN. We use production rules, because Conditional events are triggered when their conditions evaluates to true and then some other activities or events are invoked/triggered. That is, this corresponds to the behavior of production rules.

In Figure 106, we show an example of the following rule: *if customers' rental expected return date is less than current date (expected return date is passed) and the rental is still open (i.e., car has not been returned), then record bad experience on the customer and inform the customer about it*. In Figure 106a, we show this rule as am rBPMN process, where we have a Condition event that is connected to the production rule shown in Figure 106b. When this rule evaluates to true, two tasks are invoked, including the *Record Bad Experience* and *Inform Customer* tasks.



a) rBPMN model for record bad experience decision

b) Corresponding production rule

Figure 106. Integration between production rules and start Condition event

Conditional events can also be used if we want to invoke an activity, during execution of some other activity. This is possible if we attach an intermediate Conditional event on an activity edge and chose not to interrupt the activity.

In Figure 107, we show a modified business process for the rule shown in Figure 106b. In this case, during the execution of the *Car in rent* task, if the condition of the attached production rule evaluates to true (rental car is not returned until *expectedReturnDate* – see Figure 106), the process flow goes to the *Record Bad Experience* and *Inform Customer* tasks. If the rule condition does not evaluate to true, the Customer has returned the car till the expected return date (intermediate message event from the Customer), and the sequence flow continues to the complex gateway. The complex gateway is used to select the first path that arrives and to ignore the others. Therefore, the *Calculate Rental Price* activity, will be invoked when the car is returned (message event) or during the execution of the *Car in rent* task (Conditional event with production rule – Cancel case), when for example, the rental can be extra charged.

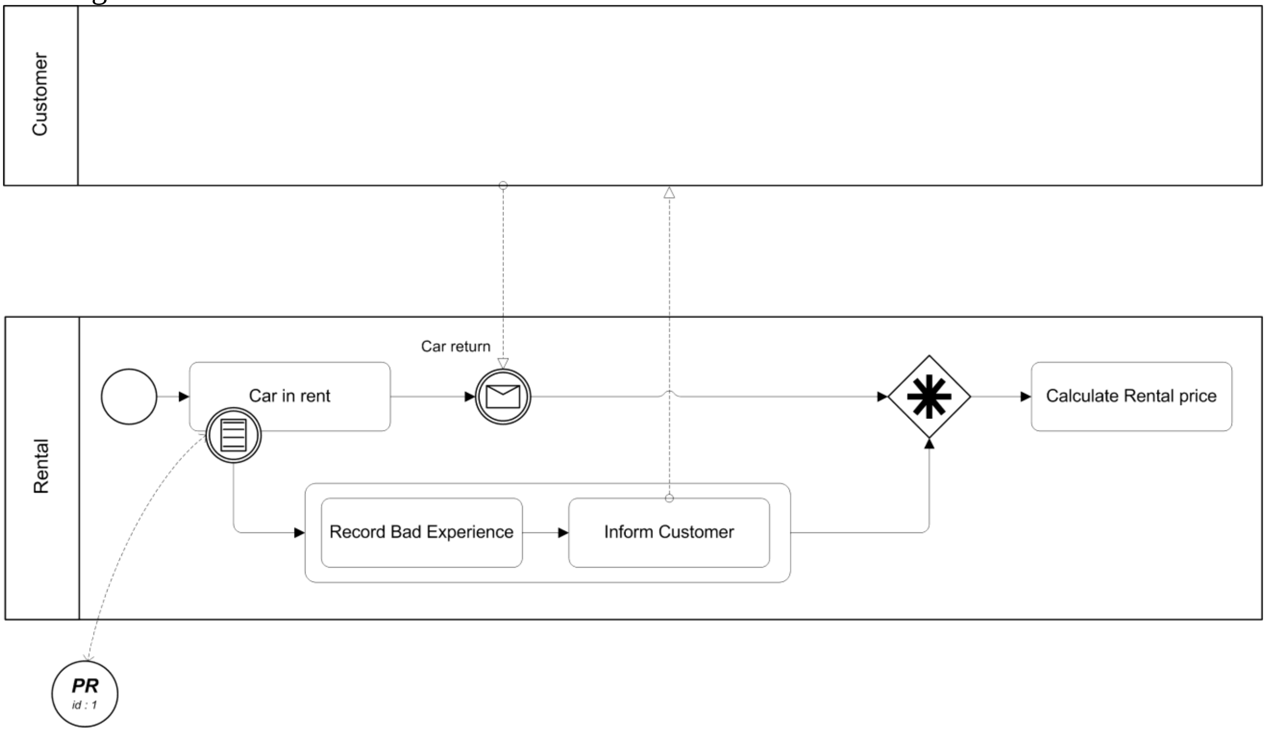


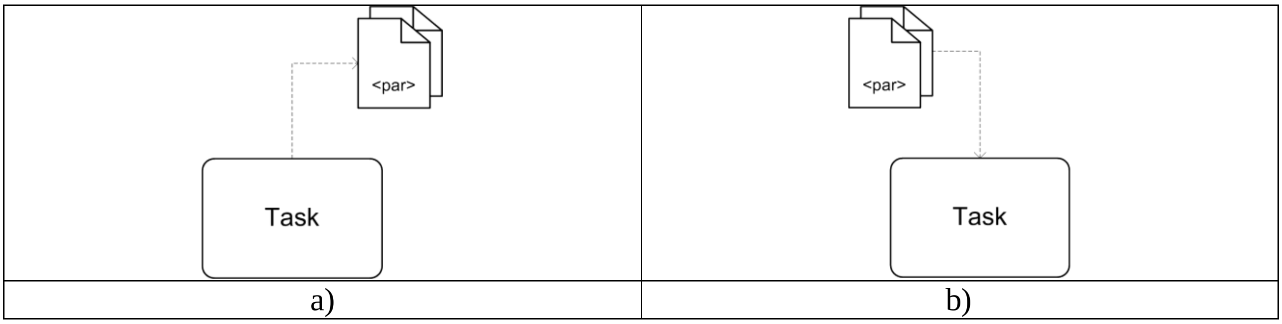
Figure 107. Integration between production rules and intermediate Condition event

3.1.4. Extensions for choreography modeling

One of the main objectives for rBPMN is to model choreographies. The standard BPMN cannot capture several choreography aspects, as recognized in [25]. Those aspects include extensions of BPMN to allow the representation of multiple participants, correlation and reference passing. In order to fully support choreography modeling, we need to take into account these aspects. We describe these aspects in the rest of the section. Those aspects are:

- **Multiplicity of participants**
  - *Problem*: In business process models, it is often needed to represent multiple participants of the same type (i.e. Pool), for example, multiple participants can involve in a conversation.
  - *Solution*: for distinguishing multiple participants from one in the pool, we will use the "Multiple-instance participant" marker (denoted with ||| in the bottom part of the pool) which is introduced in [89]. This marker means that a pool represent not one, but one or more participants.

- **References**
  - *Problem*: Following the previous problem, it is often needed to distinguish one participant from multiple participants, as we need to know, for example, which participants did some action in a process.
  - *Solution*: As introduced in [25], the main challenge with multiple participants is that we need to distinguish individual participants out of this set. The authors in [25] introduce references and reference sets as special data objects enhanced with <ref>, where a reference can be connected to a flow object via an association. Reference sets cover those cases where it is needed to select a subset of participants involved in one conversation. We base our rBPMN extension on reference sets, where we integrate both reference sets and references in one mutual concept call *participant set*, which may contain zero or more references to participants (see Figure 116). Every participant set could optionally have a name below the <par> annotation. Figure 108a shows an association from a receiving flow object to a participant set object. This association denotes that a message will be stored in the associated set. The actual participant reference in the set is represented by a participant reference object associated with a flow object. Figure 108b illustrates that an association emanating from a participant set leading to a task denotes that a message is sent to this participant. If an association leads to a receiving flow object (message event, task), a message from (a) participant(s) in this participant set is/are expected. When the participant set is associated with a multiple instance task or sub-process, this situation denotes that the loop will iterate over that participant set. We also retain references because of mapping to BPEL4Chor (see appendix B).



a) b)

Figure 108. Participant sets (graphical representation)

- **Correlation information**
    - *Problem*: Sometimes, it is not always known which participant sent a message, so that a returning message can be sent back to that participant.
    - *Solution*: Extending the abovementioned References, where an interaction partner in a participant set is certain, we propose an extension to BPMN message flows. We represent this extension by introducing an association from *Message* to *Participant* (see Figure 115). In the case when an interaction partner is not certain, the request message sent from a requester could carry the requester's identifier, which is then also contained inside the response message.

A participant set object can be associated with message flows as presented in Figure 109, or with a pool in case of an interaction modeling. This realizes link passing mobility: The associated participant objects are referenced over the message flow.

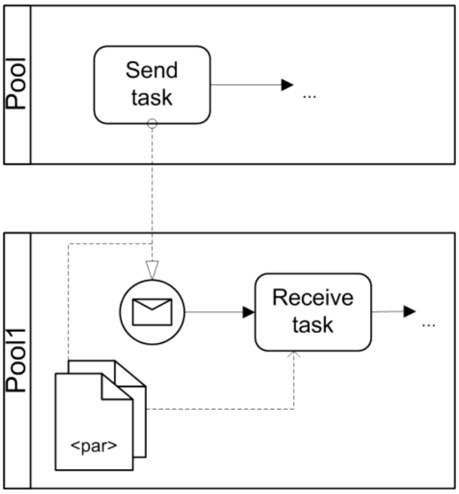


Figure 109. Passing a participant reference over the message flow

In order to create choreographies by means of interaction models, one can use the *Process* concept from the rBPMN *metamodel* or the *Choreography* concept [89]. We prefer to use the plain *Process* concept, as the Choreography concept represents a narrower type of a workflow, which contain only three elements (activities): *ChoreographyReference*, *ChoreographyTask*, and *ChoreographySubProcess*. Choreography-based languages, such as WS-CDL [55] and as it is recognized in [22], need distinction between racing choices and racing choices. The first type of choices needs a data-driven XOR-gateway, because one participant decides which path to take, and we opt for using a rule gateway in this place, because rule gateway can support data-driven XOR-gateway behavior and enrich that behavior. The second type of choices need an event-based XOR gateway, because in the case of racing choices the first occuring event inhibits others from happening, and this is depicted by the event-based XOR gateways. It is also needed to associate gateways and one of the pools in order to define who actually carries out the choice [22] (see Figure 112), as well as event annotations on messages, in order to represent start, intermediate or end of an interaction (see Figure 115). Pools are empty in interaction modeling and they are left for concrete orchestrations to implement. In section 4.2.5, we show mappings between rBPMN choreographies (interaction models) and rBPMN orchestrations (interconnection models), i.e., rBPMN interface behavior models, which represent individual views on the choreography from the side of one participant. These extensions in rBPMN are shown as an example in Figure 110.

In Figure 110, we show a simple Book loan request scenario as a choreography model. Here, we have the *send* and *receive* events for every message flow message. First, the Customer sends a book request to a Library. Then, the Library use a reaction rule attached to the rule gateway to check if the requested book is available. If the book is available, the Customer is informed about that and the Book is sent to him. Otherwise, the Customer is informed that book is not available (in both cases by using message events). If the book is already rented to a customer, then exclusive event-based gateway is used to define a scenario when a user returns the book or when book return period is expired. By using these choreography models, we only define message exchanges and rules with owning side, and no implementation is defined for any participant in a process.

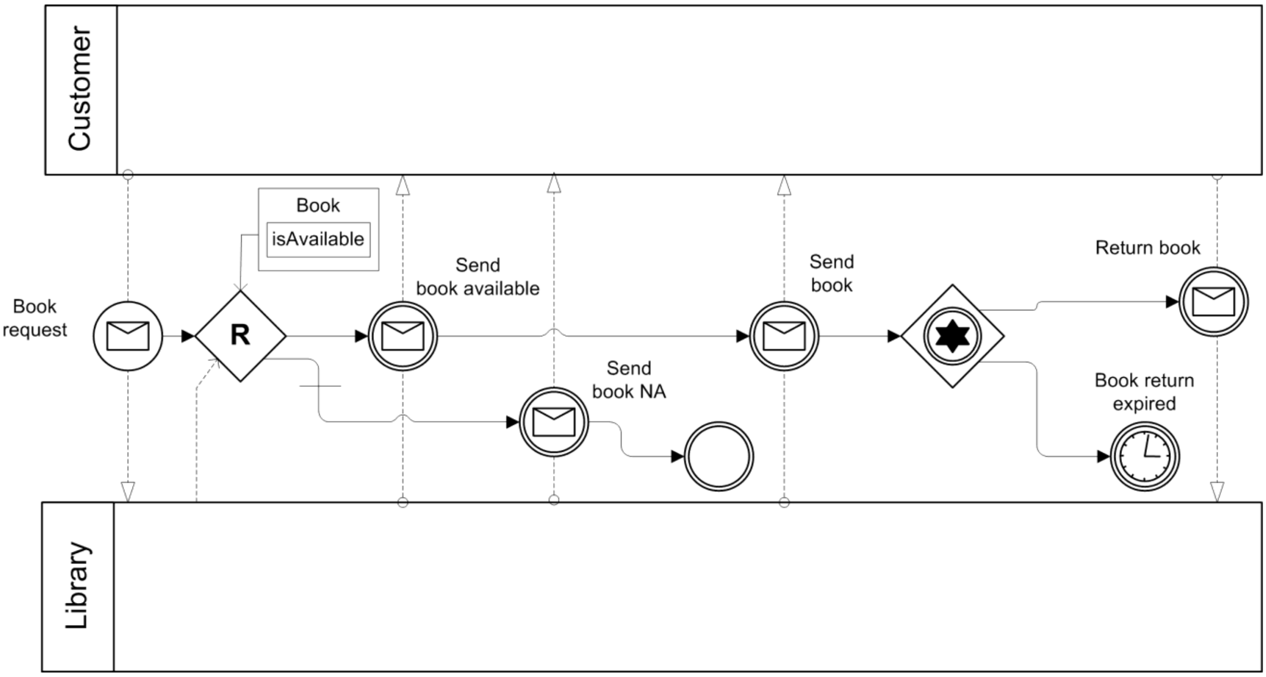


Figure 110. Library use case for interaction (choreography) modeling

3.2. Description of the rBPMN metamodel

In the MDE context, a language besides its concrete syntax (textual of graphical) needs to have an abstract syntax,too. The rBPMN language has a graphical concrete syntax shown in section 3.1. In this section, we show its abstract syntax in terms of a metamodel. As shown in the previous section (3.1), the rBPMN language consists of the BPMN and R2ML language elements. Therefore, rBPMN metamodel is an integration of the BPMN and R2ML metamodels. The BPMN metamodel is shown in Section 2.3.3.8.2, while R2ML metamodel is shown in Section 2.4.3.1.

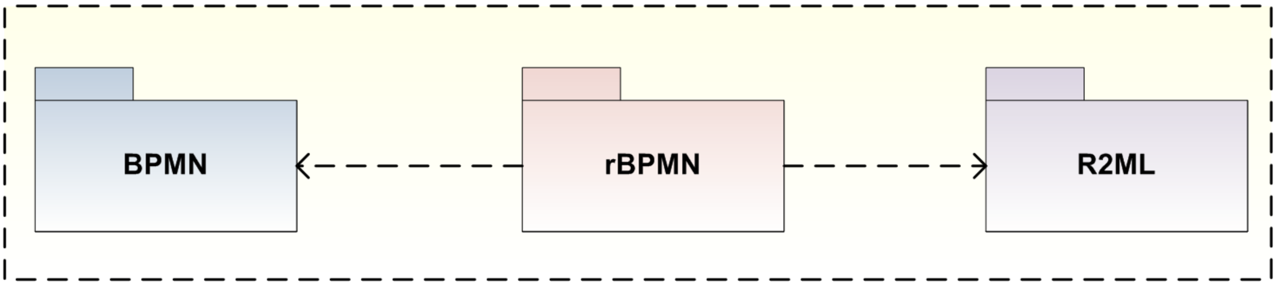


Figure 111. rBPMN metamodel packages

In this section, we present main elements of the rBPMN metamodel. The overall rBPMN picture is shown in Figure 111, where we can see that the new rBPMN package, which we introduced, includes elements from the BPMN and R2ML packages (metamodels). This means that the rBPMN metamodel introduces some new elements that may depend on the BPMN and R2ML metamodel elements, but also rBPMN users can continue using elements of the BPMN and R2ML metamodels as well.

The Basic Core package in rBPMN is shown in Figure 112. In this package, we have main elements for modeling a business process, i.e., the *Process*, *Participant (which represents Pool)* and *MessageFlow* classes. These classes are usef to model collaborations, including a collection of *Pools*, their interactions (as shown by *MessageFlows*), and *Processes*. *Collaboration* is used to describe interactions between two or more roles represented as *Participants* within *Pools*. A *Collaboration* shows message exchanged (message flows) between two or more *Participants* (*Pools*). A *Pool* represents a *Participant* in a *Collaboration*. In this package, we added the *executedBy* association between the *RuleGateway* and the *Participant*. The association is used to connect a *RuleGateway* with a *Pool* in an interaction model, in order to define which *Participant* actually carries out the choice in a choreography modeling scenario (an example is shown in Figure 110).

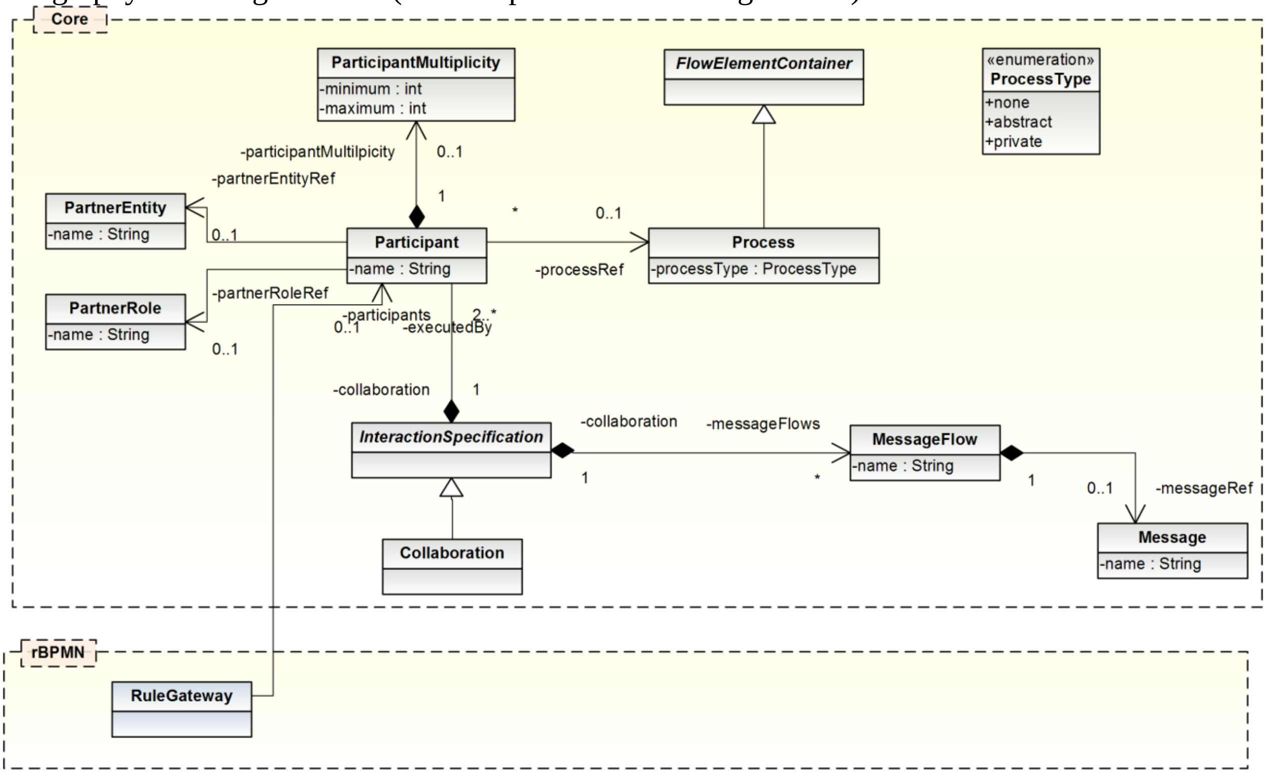


Figure 112. Core package in rBPMN metamodel

In Figure 113, we show the Process package in the rBPMN metamodel. We added the *RuleGateway* class in the Process package of the BPMN metamodel. *RuleGateway* is connected to R2ML *Rule* by using the *rule* association. In addition, *RuleGateway* inherits the BPMN *Gateway* class. In this way, we enabled that an R2ML *Rule* (that could be a *Reaction*, *Derivation*, *Production* or *Integrity* rule) can be placed into a business process as a *Gateway*, but at the same time, does not to break the R2ML *Rule* semantics. We should note here that one rule gateway could have one or more rules attached to it. In Figure 113, we can see that *RuleGateway* as a *Gateway* can be connected by using *SequenceFlow*s with other *FlowElements* such as *Tasks*, *Events* and *Gateways*. This enables us to use rules in different places in rBPMN process models. In addition, we added an association to *RuleSet* from *RuleGateway* (*ruleSet*), in order to enable for using multiple rules with one rule gateway. We represented rule conditions/conclusions by using atoms, i.e., the *ObjectClassificationAtom*, *ReferencePropertyAtom* and *PropertyAtom*. We additionaly extended the R2ML metamodel in order to support rule conditions that are defined graphically. This includes a *filter* attribute in the Atom class that is used to represent a rule condition filter (i.e., “a > 5”) for an atom used. In addition, we have introduced an association from the *AndOrNafNegFormula* to the *Rule*, because an Atom needs to contain a relation to the rule, as an *Atom* represents the condition/conclusion of a rule in a model. We introduced a *conclusionsAtom* composition to *DerivationRule*. This composition is used to denote that a derivation rule contains an *Atom* that represents its conclusion. We added a composition between BPMN *FlowElementsContainer* and*RuleSet*, so that a *Pool* or a *SubProcess* can contain *RuleSets* on a diagram/model. In BPMN metamodel, we also inherited the *FlowNode* from the R2ML *ActionEventExpression*, in order to enable an activity invokation from a R2ML rule (by using the *InvokeActionExpression*). We also added an association between the *SequenceFlow* and the *Rule*, because we wanted to choose a rule on outgoing seqeunce flows from a rule gateway, as well as between the BPMN *BusinessRuleTask* and the *Rule*. In this way, we enabled that *BusinessRuleTask* can be implemented by using R2ML concrete rules (integrity, derivation, etc.).

In order to support R2ML integrity rules, we created the *OCLInvariant* class, which are used to connect an integrity rule to the class by using an OCL invariant/constraint.

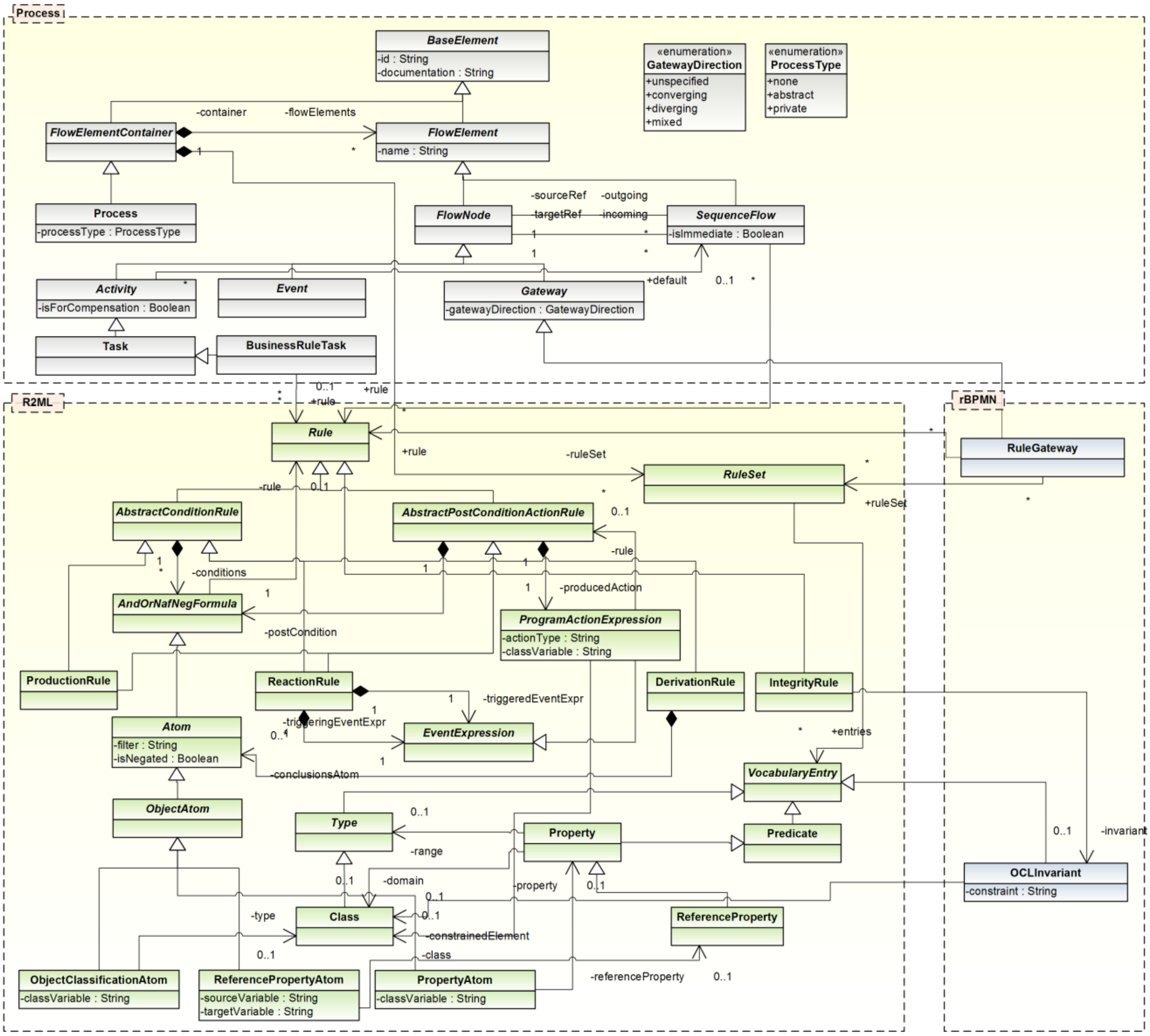


Figure 113. Process package in rBPMN metamodel

Activities are points in a process where some work is performed, and they reperent executable elements of a process. *Activity* is a *FlowElement*, and there are two main types of activites: *Tasks* (atomic activity) and SubProcess (complex activity). In Figure 114, we show extensions of the *Activity* package of the BPMN. In this package, we introduced BPMN tasks as event expressions, so that they can be triggering or triggered task of the R2ML Reaction rules (*R2MLTriggeringTask* or *R2MLTriggeredTask* class, respectively), or produced actions of R2ML Production rules. This package also contains a modeling support for subprocess as triggering/triggered expressions and production actions for R2ML Reaction and Production rules, respectively (*R2MLTriggeredSubProcess* class). Along with support for tasks and subprocesses, we also added a modeling support for events and gateways by introducing classes *R2MLTriggeringEvent*, *R2MLTriggeredEvent*, *R2MLTriggeringGateway* and *R2MLTriggeredGateway*. By introducing these concepts, we enabled that an R2ML rule attached to a rule gateway can be connected to BPMN process elements, and by doing so we assured the traceability between the process (BPMN) and rule (R2ML) modeling elements. The main advantage of this solution is that we can model parts of the business processes by using rules and rule gateways.

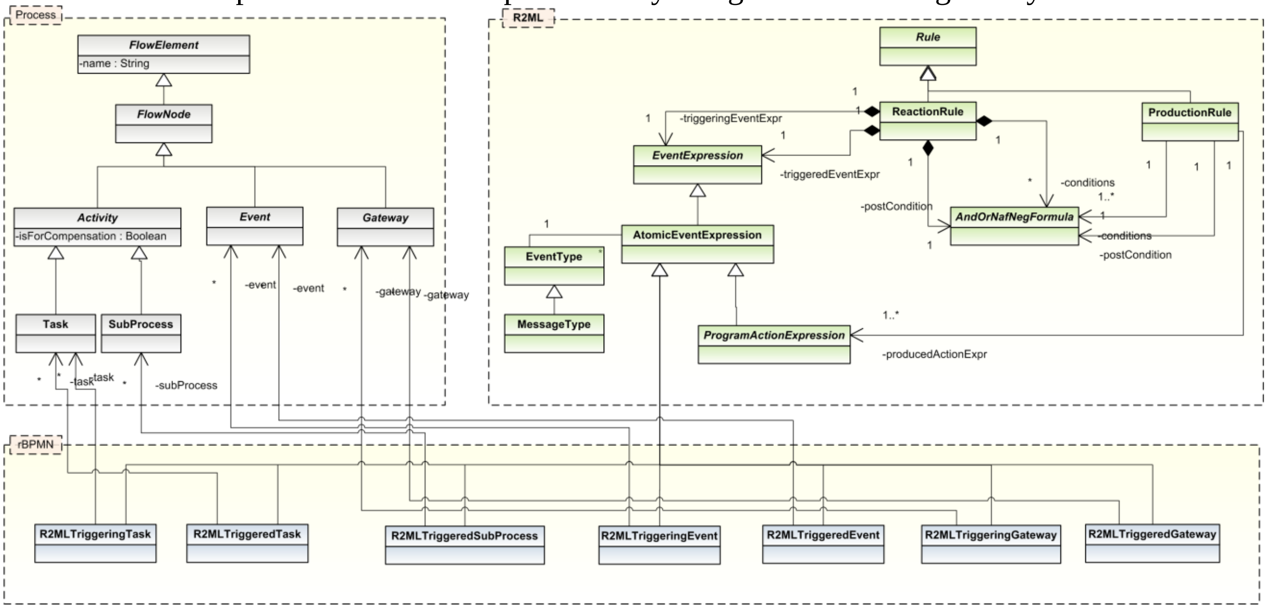


Figure 114. Activity extensions in rBPMN metamodel

In the way presented in Figure 115, we can have a rule as a valid element in a business process, but we should also have a way to connect underlying data model to the business rule. In rBPMN, we use R2ML Vocabulary as an underlying data model, so that any BPMN message can be represented with an R2ML *AtomicEventExpression* element. We enabled this by introducing the *R2MLMessageType* class, which connects to the *AtomicEventExpression* class, and the *R2MLMessageType* class is a subclass of the *ItemDefinition* class, used to define an actual structure of a message. The *AtomicEventExpression* element is connected to *MessageType*, which is used to define actual message types. We additionally attached an OCL constraint to the *R2MLMessageType* class, so that we have the same *MessageType* connected to the same *R2MLMessageType*, through the connection with the *AtomicEventExpression*. Additionaly, we introduced the *MessageEventAnnotation* class, which we use to annotate a *MessageFlow* with an *Event* in an interaction model.

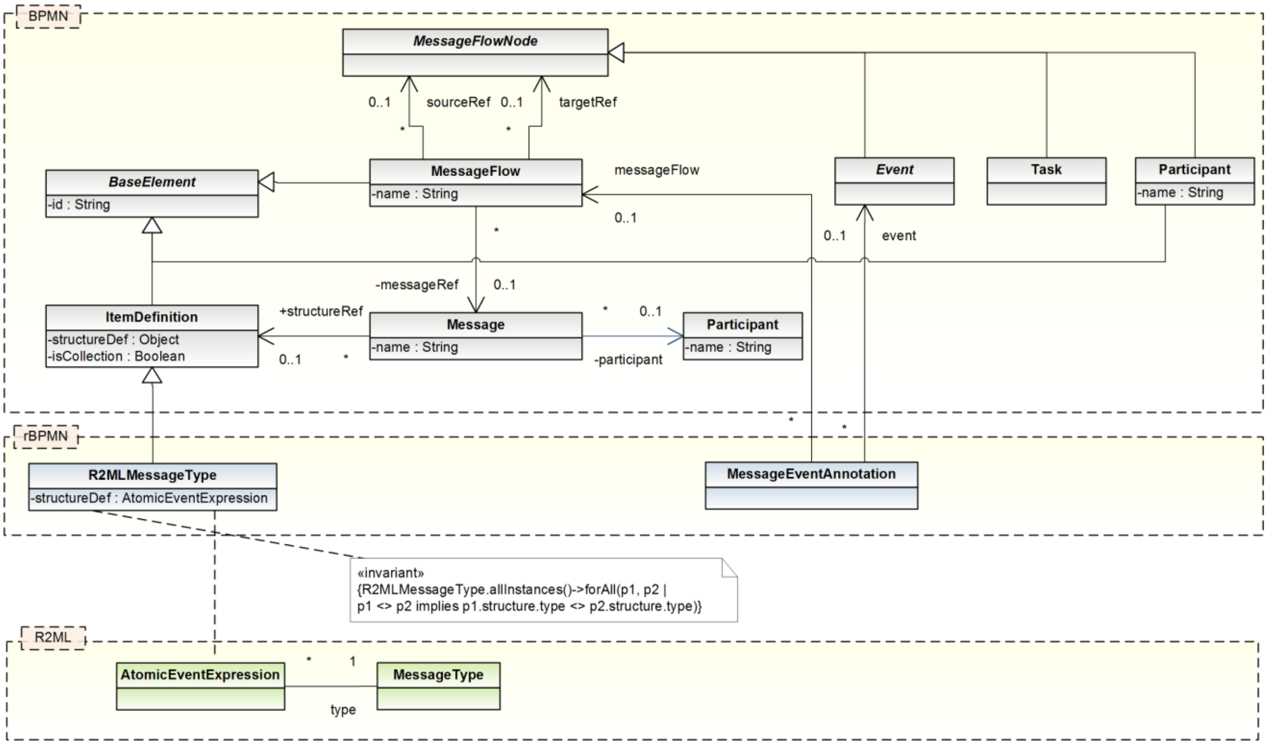


Figure 115. rBPMN data model

In the rBPMN metamodel, the *ItemDefinition* element is used to specify a *Message* structure. *Message* is connected to *ItemDefinition* through the *structureDef* relation, i.e., a *Message* can have exactly one *ItemDefinition*. We extended these BPMN elements by inheriting *ItemDefintion* and we defined its subclass *R2MLMessageType* that can have R2ML *AtomicEventExpression*, through its *structure* attribute. In this way, rBPMN messages can directly be mapped to the rules' triggering events or triggered event expressions.

In order to support modeling participant sets in rBPMN, used in interaction modeling to handle references to participants (see section 4.2), we added the *ParticipantReferenceSet* class (see Figure 116) as an *FlowElement* and *ItemAwareElement* (in order to be used as a source or target of a *DataAssociation*). We added an additional reference to *ParticipantReferenceSet* from R2ML *ObjectClassificationAtom*, so that we can use *ObjectClassificationAtom* in rule conditions.

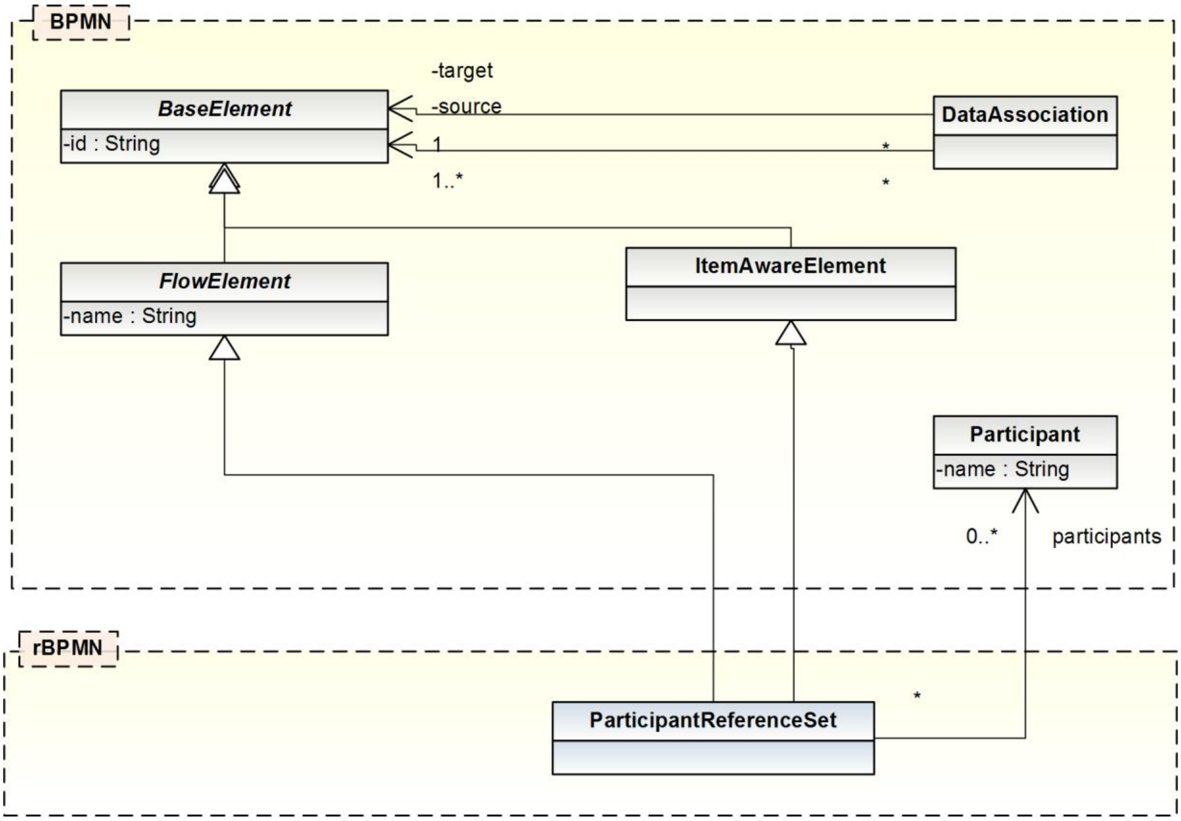


Figure 116. ParticipantSet in rBPMN metamodel

We also supported integration of R2ML production rules with BPMN *ConditionEventDefinition*, in order to support modeling scenarios as shown in Figure 106 and Figure 107. In Figure 117, we introduce an association between the BPMN *CatchEvent* class and the R2ML *Rule* class. This enables us to define a condition on a *Condition* event, in terms of R2ML production rules in the case when a *CatchEvent* has a *ConditionalEventDefinition* for its *eventDefinitions* (this is actually defined as an OCL invariant in Figure 117).

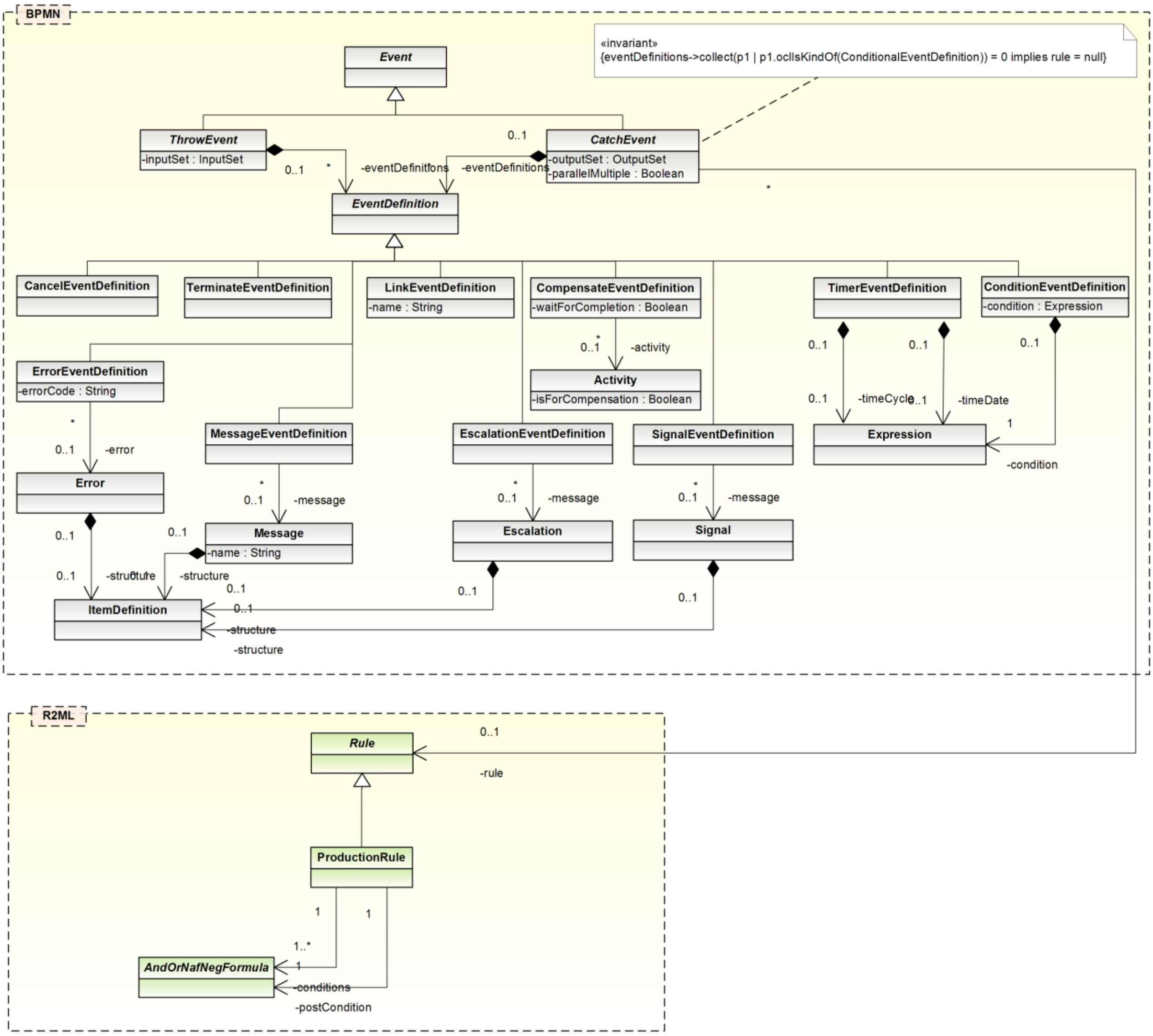


Figure 117. RuleEvent in rBPMN metamodel

In order to support the connection and use of PML in rBPMN, we introduced a concept (class) named *PolicyAnnotation* (see Figure 118) in the rBPMN metamodel. By using this concept, we can attach *PolicyAnnotation* which has a reference to the R2ML Derivation rule, to the *FlowElement* (i.e., its subclass *Task*), *Pool* and *MessageFlow*. In addition, we introduced the *PolicyAnnotationType* element as the attribute of the *PolicyAnnotation*, in order to distinguish between diffrent policy types.

WS-Policy uses a WS-PolicyAttachment to attach its policies (expressions) to WSDL definition. WSDL definitions are used because BPMN 2.0 (and rBPMN) models are mapped to the BPEL/WSDL definitions in BPMN 2.0 specification [89]. The common policy attachment points in WSDL are (distinct policy subjects): *service*, *portType* and *message* elements. The Service policy subject corresponds to the service element in WSDL document. This means that a policy attached to the service will apply to messages associated with the port in the particular service (this affects all messages in WSDL document). A port type policy subject defines operations that include message description, while a Message policy subject allows for attaching a message description with a policy expression. By using these attachment points, we ensure that a services, portTypes and messages are used only when a policy attached to them is satisfied. This enables us to have more secure rBPMN business processes.

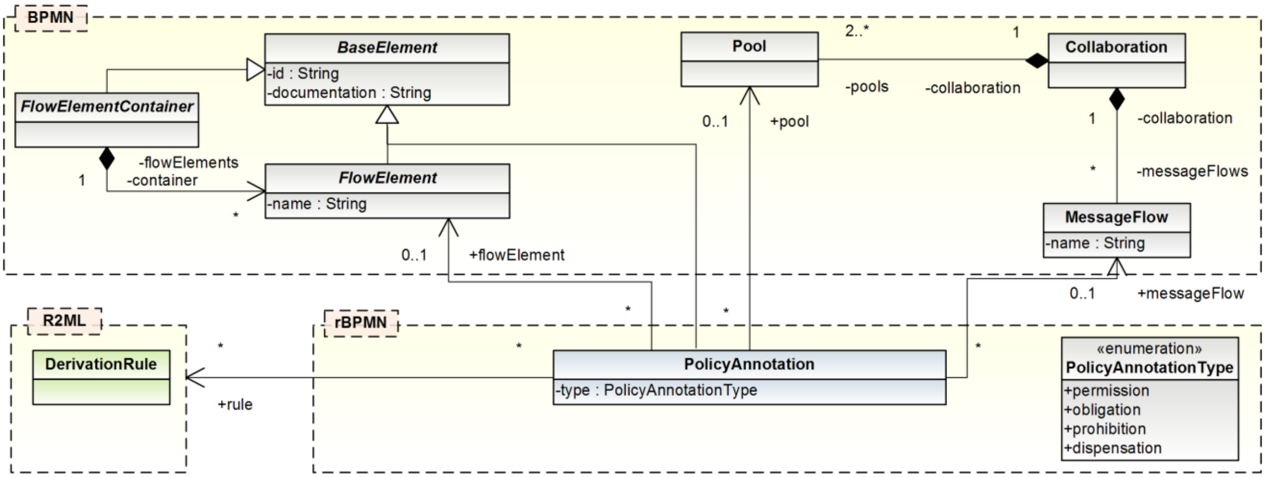


Figure 118. Support for PML policies in the rBPMN metamodel

3.2.1. Well-formedness Rules

In this section, we show (main) well-formedness rules of the rBPMN metamodel expressed in the OCL language. The metaclasses defined in the metamodel have the following well-formedness rules.

1) If a rule gateway has an event in its sequence flow just before it in a process, then a rule gateway must have at least one reaction rule attached to it.

```
      context RuleGateway
      inv: let sequenceFlow : SequenceFlow.allInstances()->select(c | c.targetRef = 
this)->asSequence()->first() in
           sequenceFlow.sourceRef.oclIsTypeOf(Event) implies
               this.rule->exists(e | e.oclIsKindOf(ReactionRule))
```

2) A Rule gateway, which has two outgoing sequence flows, must have excatly two rules attached to it.

```
context RuleGateway
inv: let sequenceFlows : SequenceFlow.allInstances()->select(c |
                            c.sourceRef = this)->asSequence()-> in
     sequenceFlow->size() = 2 implies implies this.rule->size = 2
```

3) A Reaction rule attached to a rule gateway, which has a task for its *triggeredEvent*, implies that a rule gateway, to which this reaction rule is attached, must be followed with this task in a sequence flow.

```
context RuleGateway
inv: let rule : ReactionRule.allInstances()->select(c |
         this.rule->includes(c))->asSequence()->first() in
            let sequenceFlow : SequenceFlow.allInstances()->select(c |
                   c.sourceRef = this)->asSequence()->first() in
                      rule.triggeredEvent.oclIsTypeOf(R2MLTriggeredTask)
                           implies
                                sequenceFlow.targetRef.oclIsTypeOf(Task)
```

4) Only one integrity rule is allowed to be connected to a rule gateway, so that we can have two outgoing sequence flows from such a gateway (for true and false rule evaluation results).

```
context RuleGateway
inv: this.rule->first()->oclIsTypeOf(IntegrityRule)
```

```
         implies this.rule->size() = 1
```

5) In the case of derivation rules, attached to a rule gateway, we must have at least two rule attached to it.

```
context RuleGateway
inv: this.rule->first()->oclIsTypeOf(Derivationrule)
         implies this.rule->size() >= 2
```

The other well-formedness rules for rBPMN we have defined in section 4.3 following the rBPMN definition of workflow patterns.

3.3. Integrated Methodology for development of rule-driven SOAs

In modeling of rule-ehanced business process models and taking into account the rBPMN language, presented in section 3.1, we need to introduce a method that should be used in using the rBPMN languages. In order to use rBPMN language for modeling rule-enhanced processes, we first need to define basic concepts that such language should support for modeling SOAs, and how such an approach can be used in different situations. In order to model SOA-based processes by using abstract business process languages, we need to define specific steps that should be followed by a modeler. Those steps should include decisions, such as whether some decision should be modeled in a business process directly, or by using business rules. The basic requirements of our methodology is to enable modeling of a business process with effective mechanism for updating of a business logic, i.e., in this case with business rules. This is an important issue, as the usage of business rules enables flexibility to make changes in a business process, without changing the process itself.

In Figure 119, we show our proposal for the methodology for developing rule-enabled SOAs.

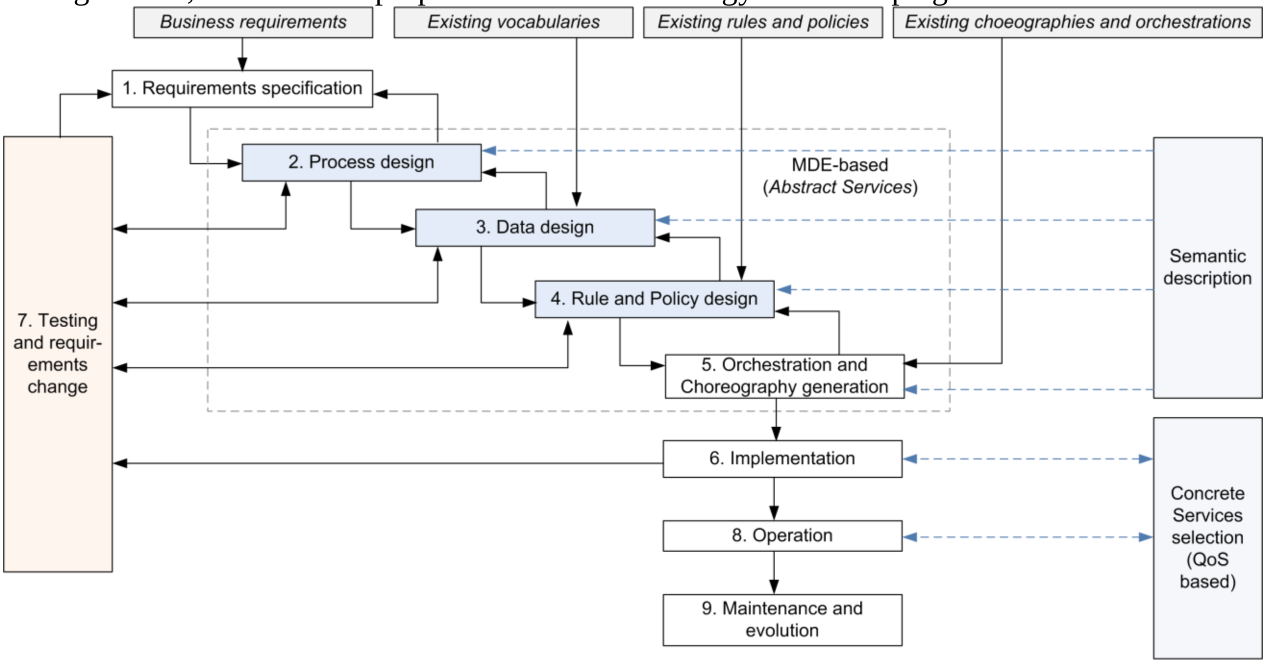

Business requirements
Existing vocabularies
Existing rules and policies
Existing choeographies and orchestrations
1. Requirements specification
2. Process design
MDE-based
(Abstract Services)
3. Data design
4. Rule and Policy design
5. Orchestration and Choreography generation
6. Implementation
7. Testing and requir-ements change
8. Operation
9. Maintenance and evolution
Semantic description
Concrete Services selection (QoS based)

Figure 119. Phases in the development process of rule and policy enabled SOAs

Our methodology is based on the MDE architecture, where development phases are applied in the iterative manner, in which various tasks are repeated until business requirements are satisfied. We use this methodology as the more agile methodologies such as the SCRUM are not suitable, because we planned to model large enterprise processes. The development phases in our methodology are:

- *Requirements specification*, in this activity, a business analyst collects information about the application domain and business functions. The output of this specification is the project initiation document.
- *Process design* is the phase of defining a general business process for the application. In this stage, using information from the initiation document from previous requirements phase, a process modeler defines an abstract business process model (by BPMN). This phase is particularly important because we need to choose a modeling approach (process or rule) by identifying points in a process that should be implemented by using rules.
- *Data design* is the activity of defining a domain model by using the information collected during the requirements specification (stage 1). This phase may include some existing vocabularies. For this

phase, we need to extend BPMN by adding it underlying data layer (i.e., some existing vocabulary language, such as UML class models [1] or R2ML vocabulary).

- *Rule and Policy design* is the activity in which rules and policies are added to the process. This activity includes creation of the rules and policies in the Rule editor. In this stage, we need to include different types of rules: integrity, derivation, reaction and production rules that can be added into process models.
- *Orchestration and choreography generation* is the activity of generation of executable orchestrations (e.g., WS-BPEL [49]) or choreographies (e.g., WS-CDL [55]) from the rule- and policy- based business processes by using a model transformation approach [1]. This step can include some existing choreographies and orchestrations, by calling their services from the modeled process. Choreographies are activities of the same process orchestration, but between activities of different process orchestrations [30], while orchestrations are modeled activities, with their relationships, that are performed within a single organization [30].
- *Implementation* includes activities from implementation generation, e.g., choreography executability on some platform, work with UDDI registry, generation of support Java application, etc.
- *Operation* is the activity of business process execution on the concrete platform.

Every activity from Figure 119, which is connected to the box “Testing and requirements change“, is going to be tested regarding to the our methodology, and such testing could lead to the requirements changes. Following phases: *Process design*, *Data design*, *Rule and Policy design*, *Orchestration and choreography generation*, could be additionally annotated with some Semantic Description (like ontologies). As the output from these phases, we get abstract services, where concrete services for those abstract services are chosen in the *Implementation* and *Operation* phases by using QoS parameters of the concrete Web services. We should also mention that our approach is completely MDE based, and this means that processes and rules are represented as instances of rBPMN metamodel (see Section 9).

As the primary objective of this dissertation is modeling of rule-enabled process models, we will describe in subsequent sections three primary steps from Figure 119 (steps 2, 3 and 4).

3.3.1. Business process design

In this step, we adapt the method from [30], in which we need to identify variable part of a process during the business process design. We adapted this method as it proposes usage of rules in variable points of process, so this makes it the most suitable solution in our case. This is the case as one our main requirements for management of changes of a business process is to isolate the parts that are changed oftenly from a static part of a business process. This method consists of two parts we describe in subsequent subsections.

3.3.2. Identification of variable segments in a process

This phase should result in variable and non-variable parts of a process. The variable points will be candidates for an implementation by using business rules. This step consists from several activities [30]:

- In the first step, we defined a process with individual activities. This is important, as we need to separate variable activities from static activities in a process, because static activities will not be implemented by using business rules.
- When a process is created, we need to identify which activities represent points of variability. Variable activites are those activities that cannot be directly executed, and they will be further specified. The static (executable) activities usually includes receive/request activities, or similar simple activities.

- When points of variability are identified, we need to estimate the level of variability, based on a several factors, as given in [30]. Those factors are:
  - *frequency of changes*: if changes are frequent, they need to be modeled by business rules, as it is easier to change separate business rules than to change the entire process at run-time. In this way, business users can change business rules, and the process changes will not affect the whole process.
  - *implementation responsibility*: implementation and modification of business rules is usually responsibility of business users, while implementation of business process is usually done by technical users. The modeling approach in this case depends on the role of the person (business or technical user) who will be responsible for the change.
  - *understanding of implications*: if an implication of a change cannot be easily understood, the scenario should be implemented by using business rules. This is the case if the effects of the change cannot be certain. Otherwise, the scenario can be modeled by the business process.
  - *source of change*: if the source of a change is like external (such as regulations), it should be modeled by a business process, as it is important to ensure that a process will be compliant to regulations and to track the event of a change. If the source of change is internal (in organization), it can be modeled by business rules, so that they can be reused.
  - *scope*: the scope defines if the impact of a business change is focused on an activity, a process or a whole organization. In the case of organization-wide changes, the modeling by business rules is suggested, as rules can be reused through the whole organization. Otherwise, if a change focuses on a single activity, only that process needs to be modified.

The decision framework that includes five given factors is shown in Table IX. We should note that this framework needs understanding of a change, as well as its context.

Table IX. Modeling approach decision framework (adapted from [30])

Business rules				**Business processes**	
Frequency	Hourly	Daily	Weekly	Monthly	Annually
Implementation Responsibility	Business User	Business Analyst	Business / System Analyst	System Analyst	Programmer
Understanding of implications	Very Low	Low	Medium	High	Very High
Source of Change	Internal	Subdivisions	Divisions	Business Partners	External Agencies
Scope	Company-Wide	Multi-Process	Process	Activity	Whithin Activity

3.3.3. Identification of appropriate software patterns

When we identified all variability points, and decided that those points are to be implemented by using business rules, we need to choose the workflow patterns needed to model concrete variability point. For example, if a variability point needs to be defined as a sequence of parallel activities, the

Parallel Split pattern should be used. If one of several activities needs to be chosen, the Exclusive choice pattern is used.

In Table X we give an overview of workflow patterns based on a common variability points that may occur in a process design. Detailed usage of rules in workflow patterns can be seen in section 5.

Table X. Usage of workflow patterns for variability points

Pattern group	Pattern	Usage of rule-enhanced pattern in a process
Basic control-flow patterns	Sequence	Activities can be given in sequence.
	Parallel Split	Activities can be given in parallel.
	Synchronization	Parallel activities needs to be joined.
	Exclusive Choice	One activity need to be chosen from several.
	Simple Merge	Activity needs to be executed when any preceding activity ends.
Advanced branching and synchronization patterns	Multi Choice	One or more activities needs to be performed.
	Multi Merge	Two or more branches needs to be converged into a single branch.
	Discriminator	Activity is started when one of the preceding (usually parallel) activities complete.
	Synchronizing Merge	Convergence of two or more branches into a single subsequent branch, where preceding activities needs to be synchronized.
Structural patterns	Arbitrary Cycles	Section of a process needs to be repeated.
	Implicit Termination	Process should be terminated when there are no remaining activities to be completed.
Multiple Instances patterns	MI without synchronization	Multiple instances of an activity need to be created. These instances are independent of each other and run concurrently.
	MI with a Priori Design Time Knowledge	Many instances of one activity are needed to be created, and the number of instances is known at design time.
	MI with a Priori Runtime Knowledge	Many instances of one activity are needed to be created, and the number of instances is variable.
	MI without a Priori Runtime Knowledge	Many instances of one activity are needed to be created, however the number of instances is not known at design-time, nor it is known at any stage during runtime, until immediately before the instances of that activity type need to be created.
State-based patterns	Deferred Choice	One of several possible branches needs to be chosen, where only one of the alternative activities is executed.
	Milestone	An activity is enabled until a milestone (specific state) is reached.
Cancellati on patterns	Cancel Activity	Activity needs to be cancelled.
	Cancel Case	Business process instance need to be cancelled.

Along with the workflow patterns, proposed in the original method from [30], we adapt this method by introducing a new group of patterns, business rules patterns for agile business processes for advanced dynamicity of a process (see section 4.4) and Service interaction patterns, which are used during modeling of choreographies (see section 4.2). Business rules patterns for agile business processes include integration of derivation, constraint and process rules. By using these patterns, we enable improved dynamicity in orchestrations. An example of using patterns for agile business processes in modeling of business processes (orchestrations) in rBPMN language can be found in section 5.3. On the other side, Service interaction patterns describe simple and complex message exchange in choreography, and these patterns can be used to improve choreography modeling. An example of using Service interaction patterns in modeling of business processes (choreographies) in rBPMN language can be found in section 5.2.

3.3.4. Data design

Data represent an important aspect in a business process design, as business process activities operates on data. Data dependencies between activities in a business process are represented by data flow, where each activity has a set of input and output (data) parameters. The input data parameters came as the input for the activity at its start, and the output parameters are generated at the activity end.

A data flow in a business process is represented by a data flow graphical constructs, so that data can be visualized and validated. Data flow represents a transfer of data between activities. In a business process, a data flow is represented by messages that annotate a message flow, which brings messages as activity parameters. In this step, we need to create a common vocabulary (among rules and a process) of key terms that will be used in the next step of our methodology (rule design). The common vocabulary includes naming of all business objects (facts), constraints (attribute values, cardinalities), as well as relationships between those objects.

A common vocabulary, in our methodology, is defined by using R2ML vocabulary concepts (see 2.4.3.1.5). In such a common vocabulary, all entity types and their properties (represented by attributes of entity types) are defined graphically. Such entity types can be used to annotate data objects and message flows in a business process model, as well as to be used in a rule specification (see section 3.3.5). Message flows are annotated with messages that have entity types from the vocabulary; these entity types are in fact types of messages that they annotate. Such messages represent input parameters for activities, and they can also represent output parameters.

The common vocabulary definition is a prerequisite to the next rule design step, because rules use vocabulary concepts in their definition (for conclusion, condition and action parts). Once defined, the common vocabulary (or external vocabulary) can be used in different business processes, along with rules in rulesets that references those vocabularies.

3.3.5. Rule and policy design

After workflow patterns are identified for a concrete scenario and the common vocabulary have been defined, we can define the rules to implement workflow patterns in order to model the process.

First, based on a chosen workflow pattern (see section 4.3.), we chose the rule type and model the business rules (reaction, production, integrity and derivation rules) by using the URML notation. During rule design, we use the defined common vocabulary in order to define rule actions and conditions/conclusions. Since we define integrity rules as OCL constraints on vocabulary concepts, during the common vocabulary design, we can attach those constraints to the designed concepts. In addition, the rule design must include the workflow pattern context (number of input and output flows, multiple instances of an activity, etc.). In this step, we can include rules defined in some other language, by transforming them to the R2ML language.

Second, when rulesets are created, we model the concrete business process, where we introduce a rule gateway, and connect it to the created rules. In this step, we model the rest of the business process, which is not modeled by rules. When the rules are integrated into a business process by employing rule gateways, we need to annotate the business process with messages, which are used in the rule design, from the common vocabulary. If a rule need to include some activities (subprocesses) in its triggering or triggered event parts, the rule can be modified in order to include those activities or subprocesses.

When the process is defined and all errors have been corrected, the process can be deployed on a business process engine (extended with a rule engine), or it can be translated into executable WS-BPEL orchestration or choreography. During the process lifetime, the business process will be changed based on changing requirements, and both business process parts (static and rule-enhanced) will influenced by this changes. If a change affects only the business rules, we only need to change those rules or to adapt them to support new requirements. On the other hand, if a change affects the static part of a business process, we need to decide whether that process part should be changed, or it should be considered as a variability point and modeled by business rules [30]. This method enables us to make the process more efficient, as the process can be adapted more flexible on changes that occur.

Along with business rules, we can also model policies by using the PML language, which are represented by extended derivation rules, as shown in Section 2.4.3.3. Those policies can be then attached to different parts of a process, and then transformed into annotated executable service compositions by using model transformations between policies and concrete policy languages [56]. Details about modeling common vocabulary, as well as rules and policies in a business process, are given in section 5.

3.4. Mapping of rBPMN to service execution language

In this chapter, we describe our proposal for mappings of the main elements of the rBPMN language into service composition languages (i.e., BPEL [49] and BPEL4Chor [23]). We also present a new framework for integration of rules into a service execution engine. We choose to use BPEL, as it is capable to represent service orchestrations (interconnection models), while its extension, BPEL4Chor introduces additional constructs to represent choreographies (interaction models) with internal behavior [61]. In addition, we propose architecture for integration of rules into the BPEL engine.

3.4.1. Mapping rBPMN constructs to orchestrations (WS-BPEL)

Mappings of the rBPMN constructs follow the mappings of the BPMN to WS-BPEL language described in [89], as the rBPMN language extends the BPMN language from the same proposal [89]. In this section, we first give short a introduction of the basic mappings between BPMN and BPEL from the BPMN proposal specification [89]. Second, we show how new elements, which we introduced in rBPMN, are mapped into WS-BPEL elements and what software architecture is needed to support this mapping. Third, we describe the framework (infrastructure) for the integration of business rules into a BPEL-based process.

Basic mappings between between BPMN 2.0 and WS-BPEL are show in appendix A, and theese mappings are defined in the OMG BPMN 2.0 specification [89].

3.4.1.1. Integration of rules into BPEL

Integration of rules into service execution languages/engines is a complex task, because both, service execution languages and rule languages are based on different paradigms. There are two main approaches for integration of these languages: (1) a tightly-coupled approach and (2) a loosely-coupled approach, as described in section 2.5. We decided to follow a loosely-coupled approach, as the tightly-coupled approach has some drawbacks as reported in [113]. The drawbacks of the tightly-coupled approach are caused by the idea that an orchestration engine needs to communicate with a rule engine. However, the BPEL speicfication does not propose any BPEL API that will enable for accessing the BPEL engine. In addition, it is convinient to provide business rules as a service, as the same rules can be reused in different business processes.

We decided to map rule gateways by using BPEL <*invoke*> activities, based on the solution from [42], which is used to invoke a Rule service. In an <*invoke*> activity's *outputVariable*, we will have a rule decision, which is usually some boolean value. Based on this value, we can define different source links for a *invoke* activity, where each source link will be constrained by a *transitionCondition*, which is directly mapped from the rule gateways' outgoing sequence flow conditions (rules).

In Figure 120, we show an example, in which the Rule service is called directly. In the example, both the Port Type and the Partner Link reference the RuleService and sources link transition conditions (Link1 and Link2) of the <*invoke*> activity are based on the return value (*ruleDecision* output variable) from the *checkRule* operation invocation. Each of the outgoing sequence flows from the rule gateway is mapped to separate link elements. Based on the name of the invoke activity, the corresponding rule is invoked by the RuleService. The incoming sequence flow to the rule gateway is mapped to a <*target*> part of the <*invoke*> BPEL activity. The input data for a rule are passed through the *inputVariable* attribute of the <*invoke*> activity, while the rule output is returned through the *outputVariable* attribute, of the same <*invoke*> activity. The *inputVariable* value of the <*invoke*> activity is mapped to a rule input variable, while *outputVariable* of the <*invoke*> activity is mapped to a rule output variable, or a rule decision (boolean), in the case of integrity rules. In the case of triggering event expressions of the reaction and production rules, those expressions are mapped together with rule conditions to the corresponding *source* and *transitionCondition* of the <*invoke*> activity. If a rule evaluates to true, *source* will contain activity referenced by a rule. To alter the decision logic, it is needed to change the Port Type, the Partner Link and the transition conditions as well as to redeploy the business process, if the number of decisions (outgoing flows) changes or if a rule type is changed. In the case that we need to handle a constraint violation, we will use a *catchAll* block inside the <*invoke*> activity, to call another activity that will handle the caught constraint violation.

This solution employs a graph-oriented structure of BPEL [62]. That structure is convenient for mapping from/to rBPMN, as rBPMN also has a graph-oriented structure. This means that we can directly map rBPMN tasks into BPEL activities, and BPMN sequence flows into BPEL links. The main difference between rBPMN and BPEL graph-based models is in how conditions are specified. In rBPMN they are specified by using complex or rule gateway, while in BPEL by combining expressions on control flow links and join conditions.

```
...
<invoke name=”ruleName” operation=”checkRule” partnerLink=”RuleService”
portType=”s1:RuleService” inputVariable=”ruleData”
outputVariable=”ruleDecision” ...>
   <targets>...</targets>
   <sources>
     <source linkName=”Link1”>
        <transitionCondition>
            <![CDATA[return ruleDecision.booleanValue();]]>
        </transitionCondition>
      </source>
      <source linkName=”Link2” >
         <transitionCondition>
            ...
         </transitionCondition>
      </source>
   </sources>
...
```

Figure 120. BPEL Implementation – rule gateway realized using BPEL links with transition conditions

Another way to implement a rule gateway’s conditions in WS-BPEL is to use an *if* statement, as shown in the example in Figure 121. The direct linkage to the RuleService is to decide whether a decision result is true or false. This service enables for decoupling the decision logic from the business process. However, in this case, before the <*if*> activity, we also need to call the RuleService, by using the <*invoke*> activity, in order to get the value for *outputVariable*. This value can then be used in <*if*> conditions to choose appropriate activities.

This solution shown in Figure 121 employs a block-structured nature of BPEL [62]. This is important, because not all BPMN graph-structures constructs can be represented in BPEL, such as loops (i.e., arbitrary cycles [62]). This means that in some cases, BPEL is constrained to its block-structured constructs.

```
<invoke name=”ruleName” operation=”checkRule” partnerLink=”RuleService”
portType=”s1:RuleService” inputVariable=”ruleData”
outputVariable=”ruleDecision” .../>
...
<if>
     <condition>
               <![CDATA[return ruleDecision.booleanValue();]]>”>
     </condition>
     ...
     <elseif>
          <condition>
          ...
          </condition>
     </elseif>
     <else>
          (optional)
     </else>
</if>
```

Figure 121. BPEL Implementation – rule gateway realized using a BPEL switch condition statement

3.4.2. Mapping rBPMN constructs to choreographies (BPEL4Chor)

In order to translate rBPMN interaction models into executable choreographies, we need to translate main rBPMN elements to elements of a choreography description language. We choose BPEL4Chor [23], because we can reuse the mapping between rBPMN and BPEL from appendix B, which includes the integration of rules into BPEL. In addition, we choose to use BPEL4Chor, as we have introduced some of its concepts in rBPMN (see section 3), in order to model choreographies successfully. A mapping of rBPMN choreography-based constructs to BPEL4Chor is then straightforward. We organized this mapping in three main parts: generation of participant types in a participant topology; creation of participant references and participant sets; and generation of message links from the message flow. These three parts represent a participant topology (structural aspect of choreography). Additionally, after defining the topology, Participant behavior descriptions (control flow and data flow) and optionally a Participant grounding (links to WSDL definitions) is needed to be defined, as described in [24].

More details about this mapping can be found in appendix B.

3.4.3. Architecture for integration of rules into BPEL

In order to use and invoke rules in a BPEL business process, it is needed to integrate a BPEL orchestration engine with a rule engine. A possible architecture for such integration is an Enterprise service bus [113], which is actually a service middleware on which a BPEL engine and a Rule Broker is connected (see Figure 122). The Rule Broker is used to unify access to different rule engines. In addition, Rule Interceptor Service is needed to bridge between rules and BPEL process, i.e., it intercepts each BPEL Web service call to apply business rules.

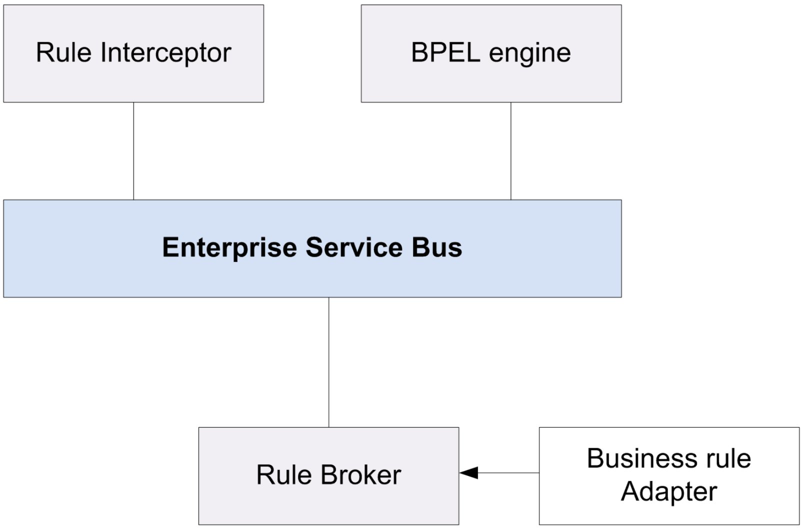


Figure 122. Service-Oriented Approach (adapted from [113])

However, this type of integration involves introducing a whole new architecture with Enterprise service bus and Rule broker. We consider a simpler solution, where rule service is provided in a form of Web services. Therefore, a rule engine may expose a WSDL interface for accessing different rules or rulesets (as shown in Figure 123). This solution enable us to change rule engines dynamically; that is, it is only needed to use the same WSDL interface for accessing rules.

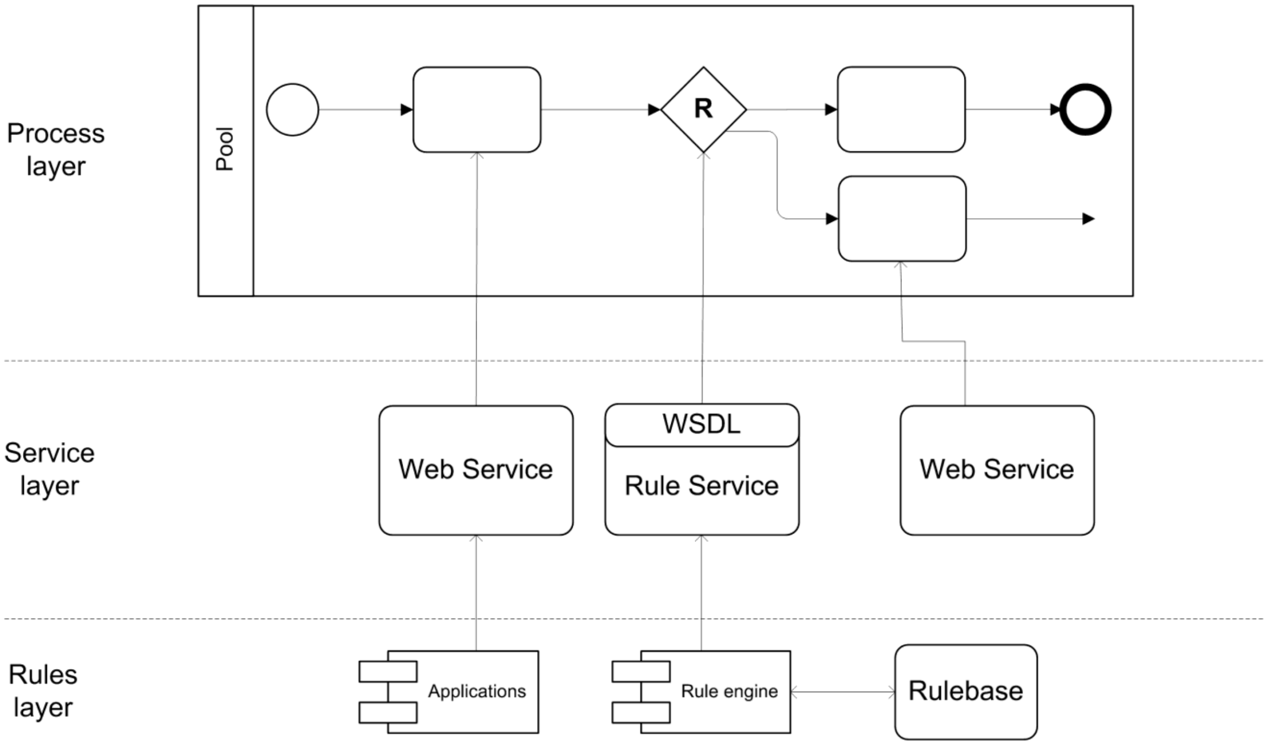


Figure 123. Business rules exposed as a service (adapted from [48])

The Rule Service can be implemented similarly to the Rule Broker [113]. The Rule Service actually represents a Web service interface to the rule engines (see Figure 124). It is modular, and by using

Business Rule Adapter (that can be realized by using Java Rule API [51]) different rule engines can be pluged-in and used. Generation of WSDL interfaces for the Rule Service is based on a rule and a rule gateway description call for a particular process. Therefore, each business rule is exposed as a service, and it can be used in different BPEL processes.

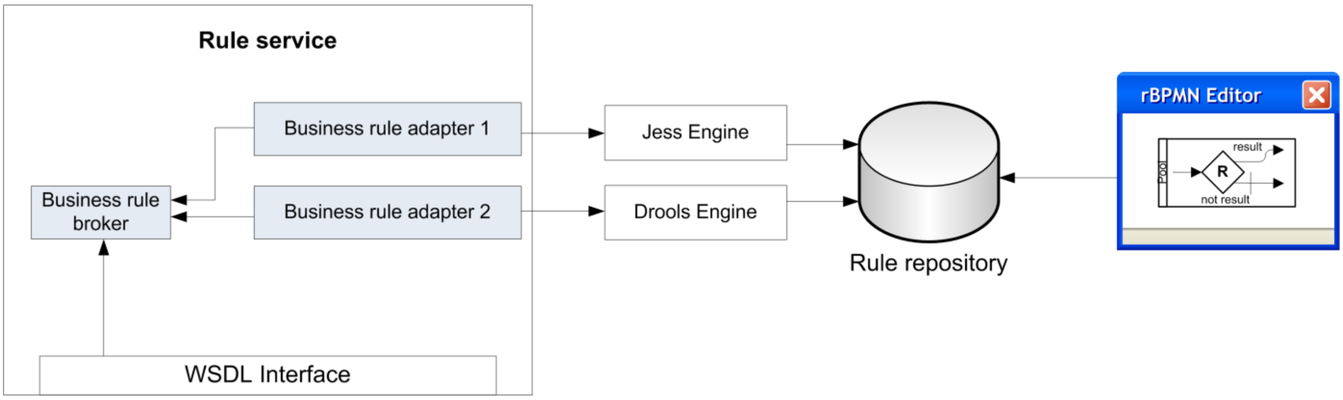


Figure 124. Rule service architecture (adapted from [113])

An example of a Business rule broker implementation is already analyzed, and its proposed prototype can be found in [114]. The proposed realization of the Rule broker is done in Java by using Java Rule Engine API [51]. The Rule broker is based on the *IRuleEngine* interface, which must be implemented by any rule engine adapter. The *IRuleEngine* interface has an *executeRules* method, which is used to execute rules in a knowledge base. The *RuleEngineProxy* is the class used to communicate with the Rule broker. It uses a *PluginManager* to load installed adapters and it keep references to these adapters, too.

4. Modeling Service and Process Patterns in the rBPMN Language

In this section, we show how the rBPMN language can be used to model Message Exchange Patterns, Service Interaction Patterns for choreography modeling and Workflow (control) flow patterns for orchestrations modeling. In these patterns, we illustrate the possible benefits from the introduction of rBPMN.

4.1. Representation of Message Exchange Patterns in rBPMN

In order to represent basic SOA concepts by using rBPMN, we will first show how the basic message exchanges in SOA can be modeled in rBPMN. Such message exchanges between Web services are typically represented by using the Message Exchange Patterns (MEP) [137]. They define what type of messages (including faults) can be exchanged and in what order. Regardless of how complex tasks performed by a Web service are, almost all of them require the exchange of multiple messages [31]. It is important to coordinate these messages in a particular sequence, so that the individual actions performed by the message are executed properly. MEPs are a set of templates that provide a group of already mapped out sequences for the exchange of messages [31]. The following MEP definition states that "[a] *MEP specifies in a reusable manner the ability of a service to receive and/or send messages. It describes the set of exchanged messages in terms of their order and multiplicity, i.e. whether a message is sent to or received from a single node or whether a message is sent to or received from multiple instances of a node. Optionally, different node types can also be identified*" [86]. This means that MEPs define how services should be used, as they can coordinate input and output messages related to a certain operation. The WSDL 2.0 specification defines three MEPs [137]:

- In-Only pattern – supports a standard fire-and-forget pattern (i.e., only one message is exchanged);
- Robust In-Only pattern – is a variation of the In-Only pattern that provides an option of sending a fault massage, as a result of possible errors generated while transmitting, or processing data;
- In-Out pattern – presents a request-response pattern where two messages (input and output) must be exchanged.

However, WSDL 2.0 specification offers the possible use of five more patterns, introduced as additional patterns [138]:

- In-Optional-Out pattern – is similar to the In-Out pattern with the following exception: sending a message that represents a response is optional, and because of this, a requester that has started a communication should not expect this message;
- Out-Only pattern – consists of exactly one message, and is most often used for message notification;
- Robust Out-Only pattern – presents a variation of the Out-Only pattern that provides an option of sending a fault massage;
- Out-In pattern – consists of exactly two messages, but in this case, a service provider is the one that initiates the exchange of messages;
- Out-Optional-In pattern – is opposite to the previous pattern, where the input message is optional.

Generally, MEPs are divided into two major groups depending on which side initiates an interaction. In-bound patterns are two initiated by a service requestor, i.e., the interaction starts with an incoming message. Out-bound patterns are imitated by a service itself, i.e., the interaction starts with an outgoing message. The rest of the section in organized into the subsections as per this MEP classification. These sections illustrate how each of the abovementioned patterns can be modeled by using rBPMN. In this section we use the case study of service-oriented distributed health information systems. Such systems require collaboration between different public and private section parties (e.g., health care establishments, insurance companies and police). The general problem of the use of BPMN in this problem has been discussed elsewhere [112]. Here, we focus on fragments of processes, MEPs and the role of business rules in such interactions.

4.1.1. WSDL 2.0 In-Bound MEPs

The in-bound patterns are patterns where the interaction is initiated by a service requestor, that is, a service first receives the message. In this section, we represent first every MEP in the standard BPMN and followed by their mapped and refined versions represented in rBPMN.

4.1.1.1. In-Only

The In-Only pattern consists of exactly one message received by a service from some other node (i.e., service requestor), where no fault message may be generated. In Figure 125, we show how the In-Only pattern modeled first in the standard BPMN and then mapped and refined in rBPMN. We represent partners in the message exchange as BPMN pools (given that BPMN pools are used for this purpose). The exchange of the message is represented with the corresponding BPMN message flow between collaborating pools. The message flow is annotated with the message (i.e., getInfoRequest). By their basic definition, each service is described by its interface consisting of an operation, which is communicated with through input and output messages. Thus, each operation is modeled through BPMN tasks (e.g., getInfo), given that BPMN tasks can send and receive messages. Finally, an rBPMN model of a Web service is following our previous work [111], where each service can be modeled by a reaction rule – input message is a triggering event, rule condition is a condition for invoking/executing the service, and the service operation is the triggered action.

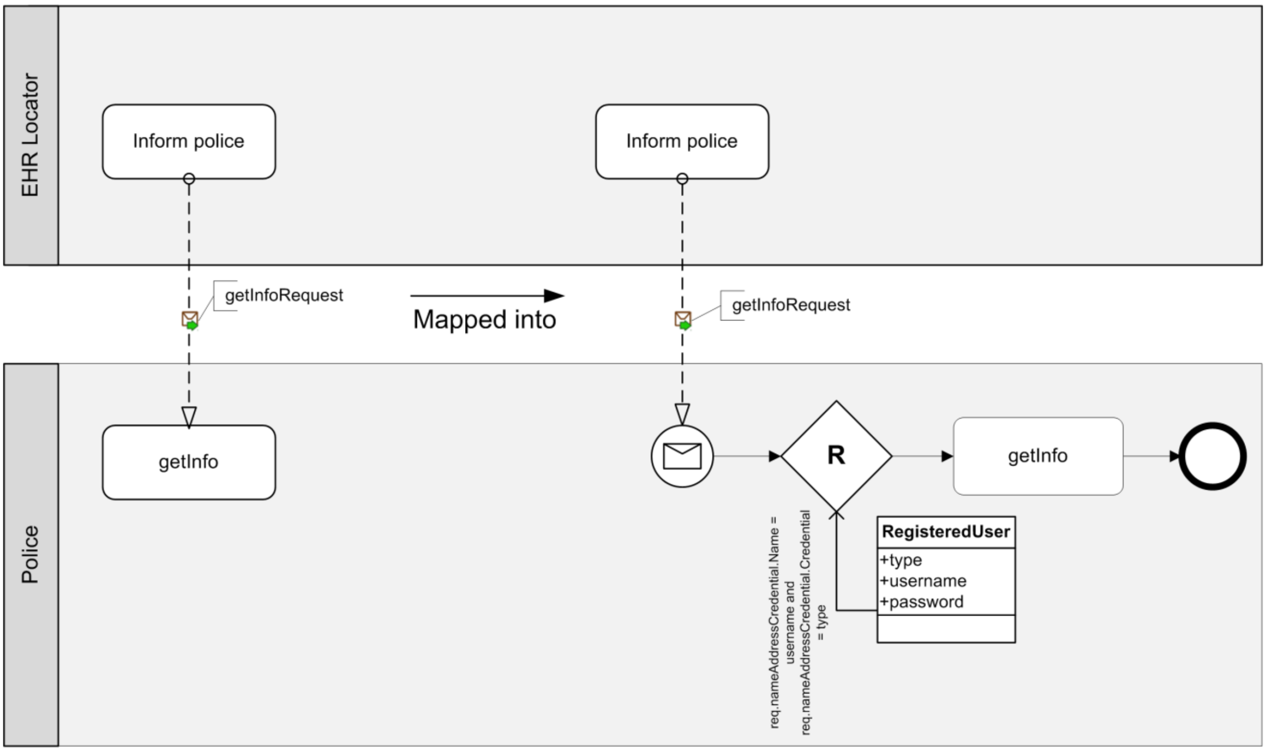


Figure 125**.** Model of the In-Only pattern in rBPMN

In the right part of Figure 125, the rBPMN "Inform police" activity sends a message "getInfoRequest" that triggers the rule gateway through a message event. The receipt of the message is represented by using a message event, as gateways in BPMN cannot receive messages (i.e., to be able to trigger our reaction rule modeling the web service). The rule gateway defines a Web service (e.g., whose operation is called "getInfo"), while the message event (received from the service client such as EHR Locator) connected to the rule gateway (i.e., the triggering event) models the input message of the service. As per our definition of the rBPMN metamodel, our rule gateway from the figure has to have a reaction rule associated with it. This establishes an explicit tractability between the rule gateway of Figure 125 and the rule's full definition represented in R2ML and graphically shown in URML. This enables to further define the rule with (post-)conditions and connect the service definition with the vocabulary elements as shown in Figure 126. Furthermore, there is an explicit traceability between the annotation of the message exchanged in Figure 125 and the message type from the reaction rule of Figure 126, which also provides the complete definition of the message.

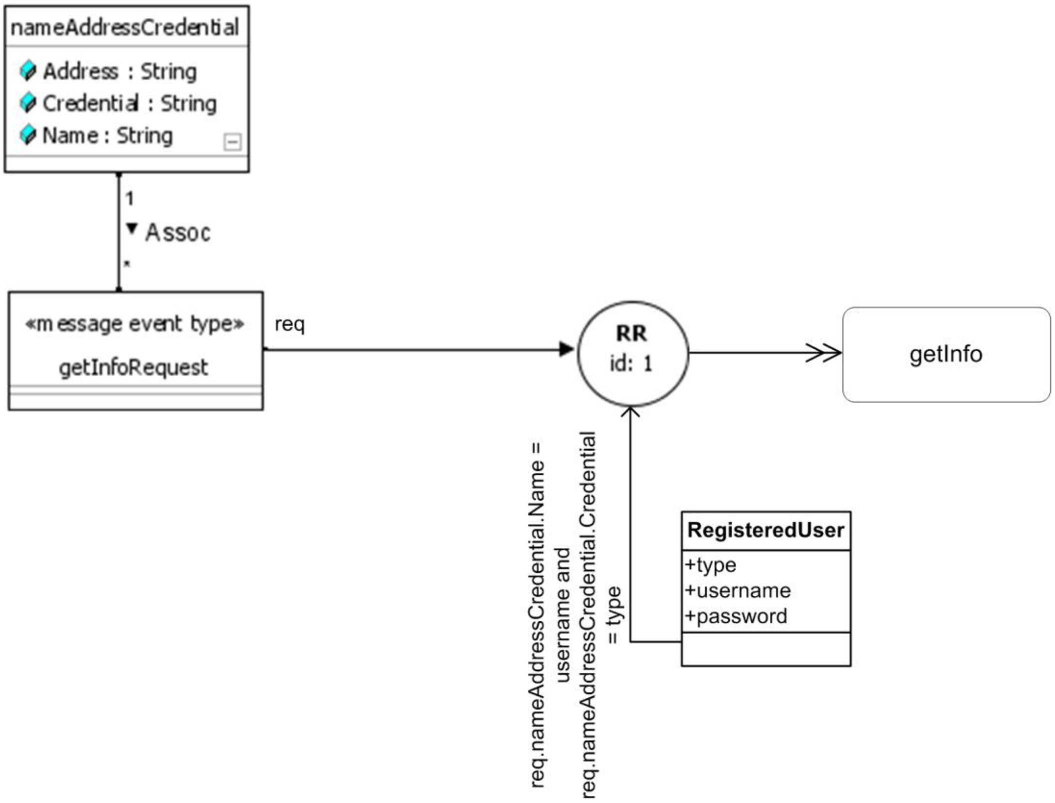


Figure 126. Associated reaction rule to the rule gateway for the rBPMN model of the In-Only pattern shown in Figure 125

In this pattern, we used a reaction rule attached to rule gateway because reaction rule has an event for its input, so when a message event happens it fires the reaction rule. In this case, a production rule cannot be used, as production rules are not triggered on an event, but rather on an true condition. The use of reaction rules is convenient because we can model an input message and output service activity. By using the rule gateway in this MEP, we add additional elements of the business logic, which overcome the definition of Web services and this pattern given in their W3c specification. Namely, this rule gateway can define a condition under which the modeled service can be used, once the input message has been received. Most importantly, this condition can be updated both at run- and design-time. For example, this condition can define from which partner service our modeled service can process requests. Another important implication of our model is that from this rule, we can generate not only the complete WSDL description, but also Java definition that encapsulates the service logic, which might generally not be visible to the service users.

4.1.1.2. Robust In-Only

This pattern can be considered a variation of In-Only. It also consists of exactly one message received by a service, but in this case, faults can be triggered by the message as specified in the *Message Triggers Fault* model. This Fault model and other fault models are described in [137]. Message Triggers Fault is a rule, which says that any message in the interaction might trigger a fault message, which must have the opposite direction from the triggering message. In the case of Robust In-Only, the input message might trigger a fault message, which has to be sent to the service requestor. In Figure 127, we show how the Robust In-Only pattern is originally modeled in the standard BPMN and then mapped and refined in rBPMN. For this pattern, we introduce the Exception handler subprocess, which is used to send the fault message to the service requestor.

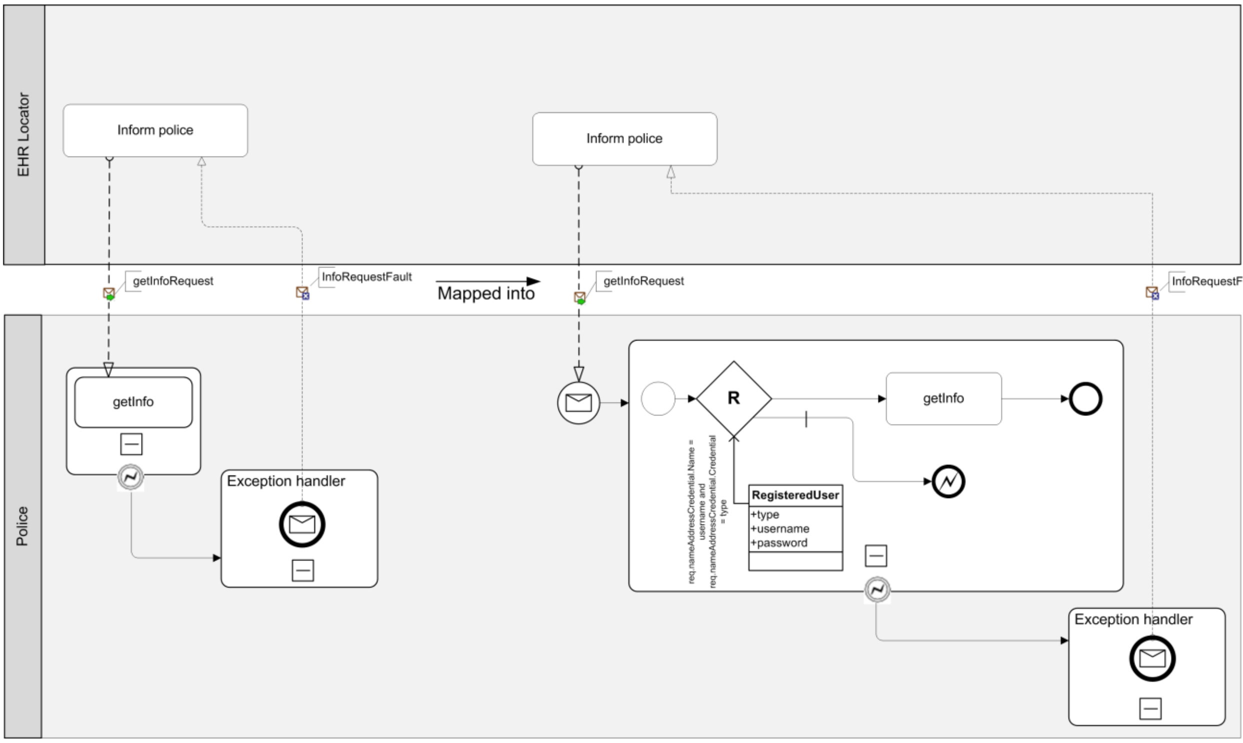


Figure 127. Model of the Robust In-Only pattern in rBPMN

In the right side of Figure 127, the input message modeled by a message event. The rule gateway, which models the web service, is placed in a subprocess. This is done in order provide a common mechanism for exception handling (i.e., Exception handler) of different types of errors that might happen in this pattern. We identify two possible types of errors for a "Message Triggers Fault" model. The first type of an error is caused by the business logic that the rule gateway models. This means that the condition of the rule gateway is not satisfied. This will result in an exception event (i.e., the else branch of the reaction rule – the outgoing arrow of the rule gateway with the backslash). Given that the ⚡ symbol is attached to the rule gateway subprocess, the generated error will be processed by the Exception handler. The handler generates the fault message (i.e., “getRequestFault”) to be returned to the service requestor . The condition of the rule gateway is to decide whether the activity “getInfo” (i.e., the service operation) should be invoked or not. The second type of error is the exception that can be raised as a result of the execution of the internal logic encapsulated into the “getInfo” task. In this case, the “getInfo” task can throw an exception, which is handled by the Exception handler. The Exception handler returns the fault message (i.e., getRequestFault) to the sender by employing the same procedure used in the first type of exception.

It is important to emphasize that by using the rule gateway, we have been able to separate fault messages that are the result of the internal service error from those that are the result of an exception of the service execution. The reaction rule connected with rule gateway from the rBPMN diagram has an equivalent rules further refined in the R2ML represented in Figure 128.

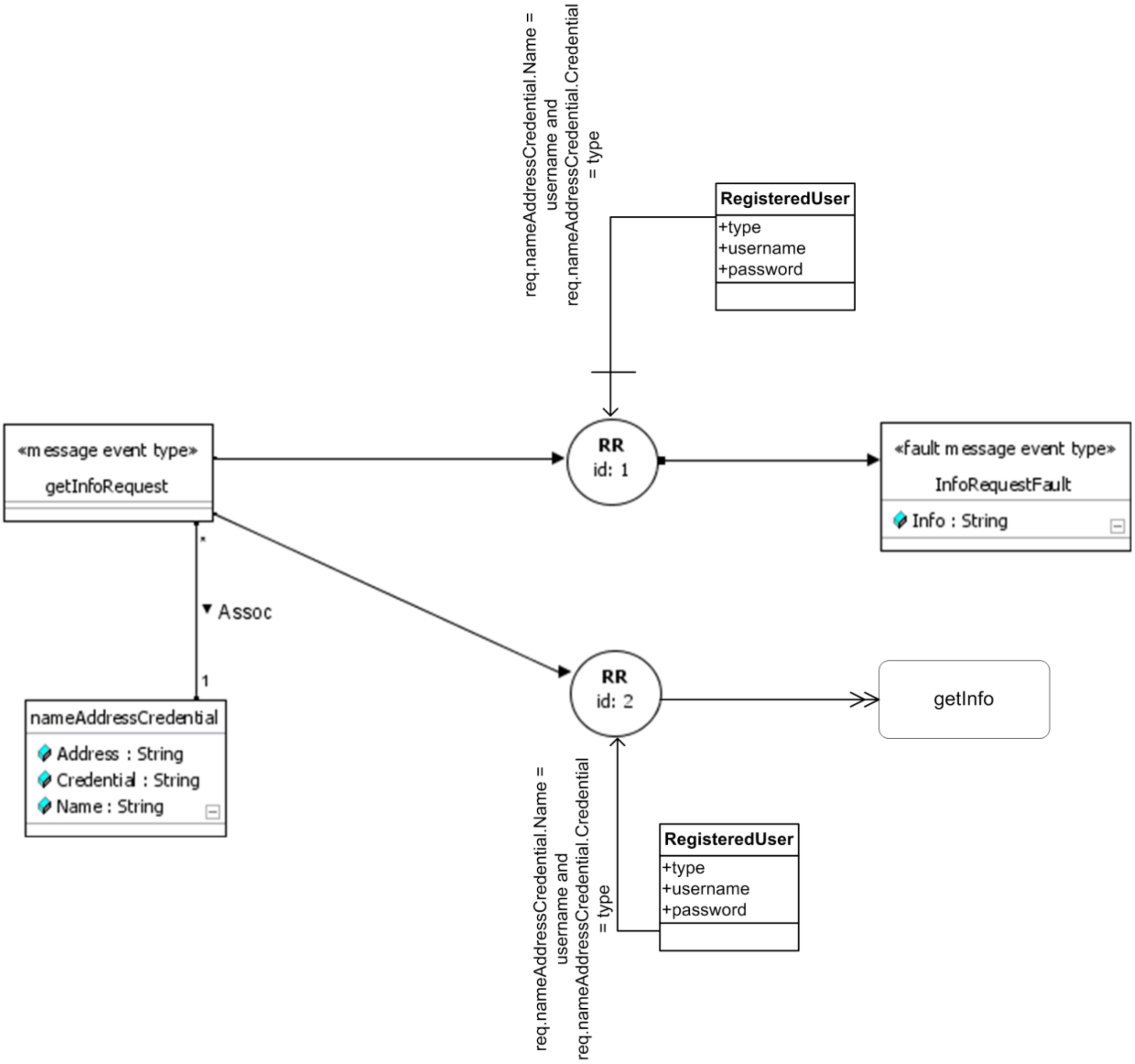


Figure 128. Associated reaction rules to the rule gateway from the rBPMN model of the Robust In-Only pattern shown in Figure 127

For the Robust In-Only pattern, we have two R2ML reaction rules that are associated to the rBPMN gateway. Both rules are triggered on the same triggering event, but the difference is in their conditions. The condition for rule 1 is the negated condition of rule 2. That is, when the condition is true, rule 2 is fired and its action (i.e., triggered event) is executed (i.e., "getInfo" task). When the condition is false, rule 1 is fired and the error event is generated. The introduction of two rules is done to avoid assuming that the implementation rule language of the modeled business logic might have support for else branches, which might cause different reasoning problems.

While translating this model into its implementation, we have two options for translating the condition of the rule gateway. One option is to implement it as a condition in the beginning of the implementation of the "getInfo" task (e.g., in the beginning of a Java method that implements the business logic of the service). Otherwise, this condition can be implemented by using a rule language. An advantage of using a rule language is the support to declarative approach, including rule externalization and share between applications, faster changes, etc. As well, we do not need to change the service logic anymore in its lifecycle even though the conditions under which the service can be changed.

Figure 128 also indicates the fully established tractability between the elements of rBPMN and R2ML. Namely, all BPMN tasks (e.g., “getInfo”) and messages (e.g., “getInfoResponse”) have their counterparts in R2ML. The tractability is established through the rBPMN metamodel. Yet, the combination of BPMN and R2ML fully complements each other (e.g., message type definition).

4.1.1.3. In-Out

This pattern consists of exactly two messages: a message received by a service (i.e., input message) from some other node, followed by a message (i.e., output message) sent to the other node. The second message may be replaced by a fault as specified in the "Fault Replace Message" model [137]. The Fault Replace Message model defines that a node, which generates a fault after receiving the input message, replaces the output message with a fault message and sends back the fault message to the sender of the input message. In Figure 129, we show the model of this pattern in both the standard BPMN and rBPMN.

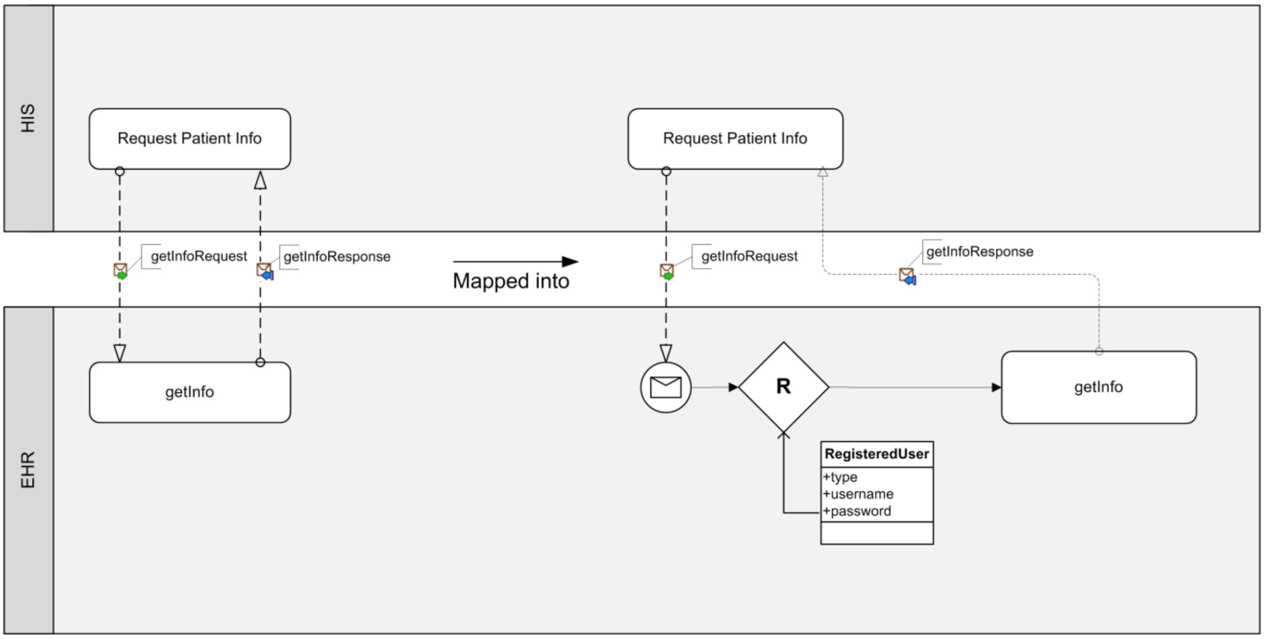


Figure 129. Model of the In-Out pattern in rBPMN

In Figure 129, the BPMN task “Request Patient Info” invokes the “getInfo” service operation by sending the “getInfoRequest” input message. The invoked operation returns the “getInfoResponse” message. The message flows between the two pools are annotated by these two messages. The rule gateway, which models the web service, accepts the input message as its triggering event. The triggered activity of this reaction rule is the BPMN task “getInfo”, which represents the execution of the service itself. In this pattern, similar to the In-Only pattern, we can define a condition of the rule gateway, which defines when the “getInfo” task can be performed.

In Figure 130, we show the reaction rule that is associated with the rule gateway of the rBPMN model from Figure 129. In this reaction rule, it is important to notice that the triggered event part of the rule is a sequence of the “getInfo” action and the “getInfoResponse” output message. This fully captures the semantics of the execution of services – the executing of the service followed by the output message.

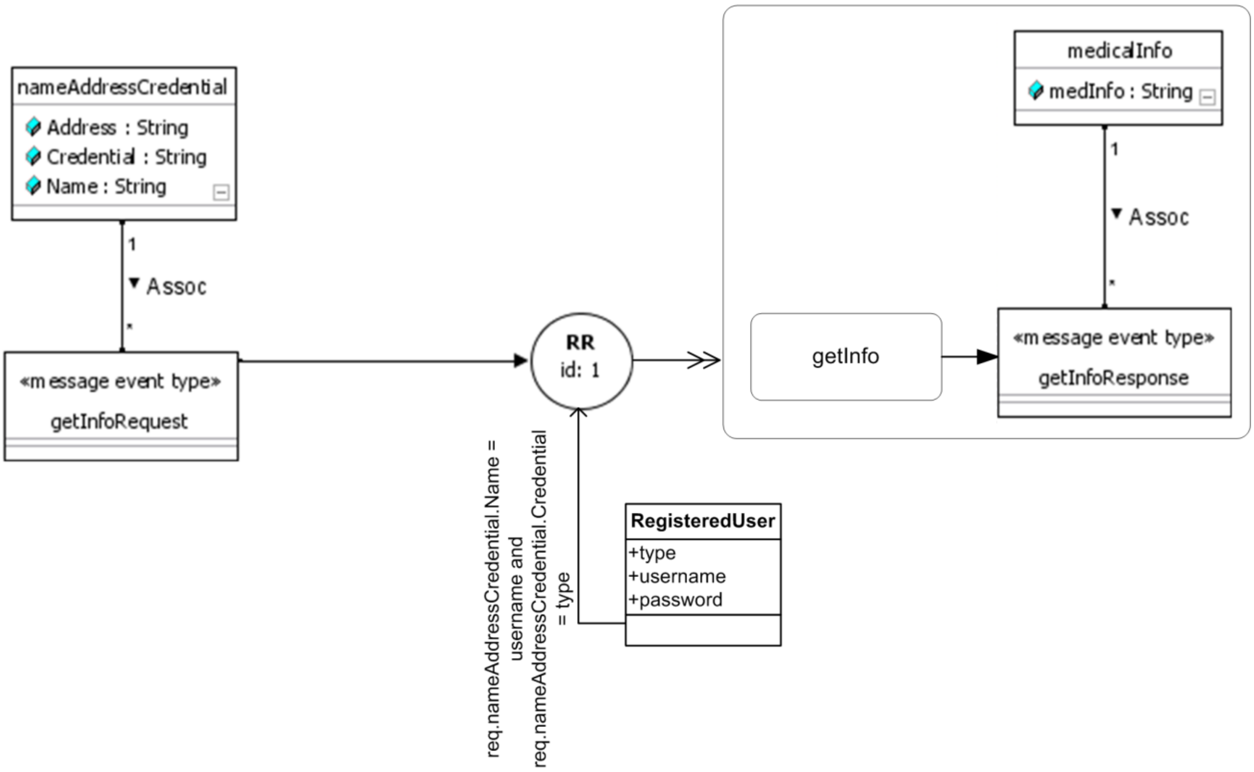


Figure 130**.** Associated reaction rule to the rule gateway from the rBPMN model of the In-Out pattern shown in Figure 129

When the service returns a fault message in this pattern, we have a similar situation to the Robust In-Only pattern. Figure 131 shows the rBPMN metamodel of the In-Out pattern where a fault message replaces the output message.

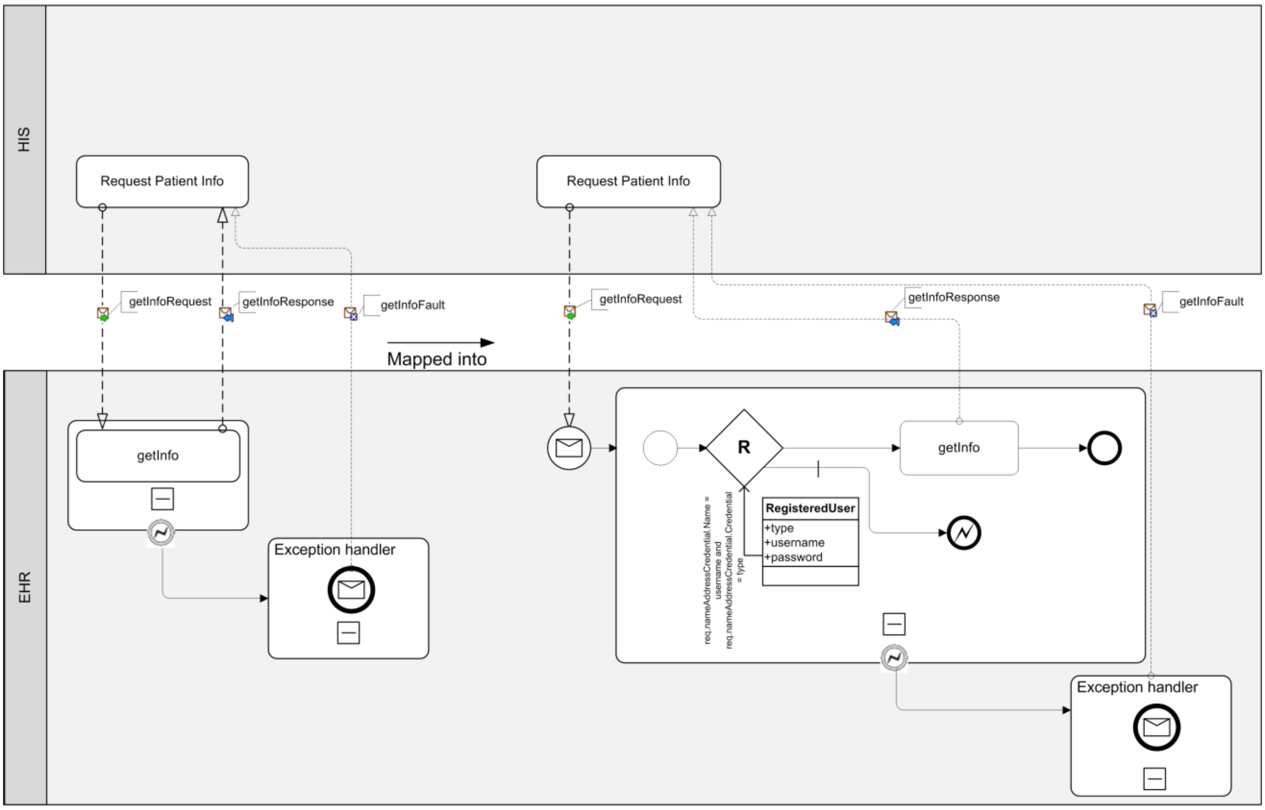


Figure 131. Model of the In-Out pattern (with fault replaces message) in rBPMN

The In-Out MEP from Figure 131 is actually the same as the one from Figure 129. The only difference is that the rule gateway can lead to the generation of an error end event (which is then handled with “Fault Handler”). This happens when the condition of the rule gateway is not satisfied. Similar to the model of the Robust In-Only pattern, we also distinguish between two type of exceptions: i) those that are generated when the condition of the rule gateway evaluates to false, and ii) those that are generated as a result of the execution of the internal service logic. When the input message is received from a sender, the Fault message can be returned to the sender in both fault cases by using Exception handler. In the first case, when the condition of the rule gateway evaluates to false, the sequence goes to the error end event in the subprocess. In the second case, an error may be generated during the “getInfo” task execution. In both cases, the Fault Handler, which is attached to the border of the subprocess, will invoke Exception handler to return the fault message to the sender. Thus, we enable exception handling in a unique way.

In Figure 132, we represent the R2ML reaction rules that are associated with the rule gateway from Figure 131. The full definitions of the conditions for rule gateway are fully specified in the reaction rules. Both rules are triggered on the same triggering event, but the difference is in their conditions. These rules means: *On patient information request, if the user is registered and provides valid credentials, retrieve the retested information and notify the user.* This pattern is similar to Robust In-Only pattern, where we also have two rules. However, the main difference is related to the case when the condition evaluates to true (i.e., rule 2 is triggered), then the sequence of the “getInfo” task and the “getInfoResponse” message is performed. In the case of the Robust In-Only pattern, we have only the “getInfo” task, since the output message is not generated.

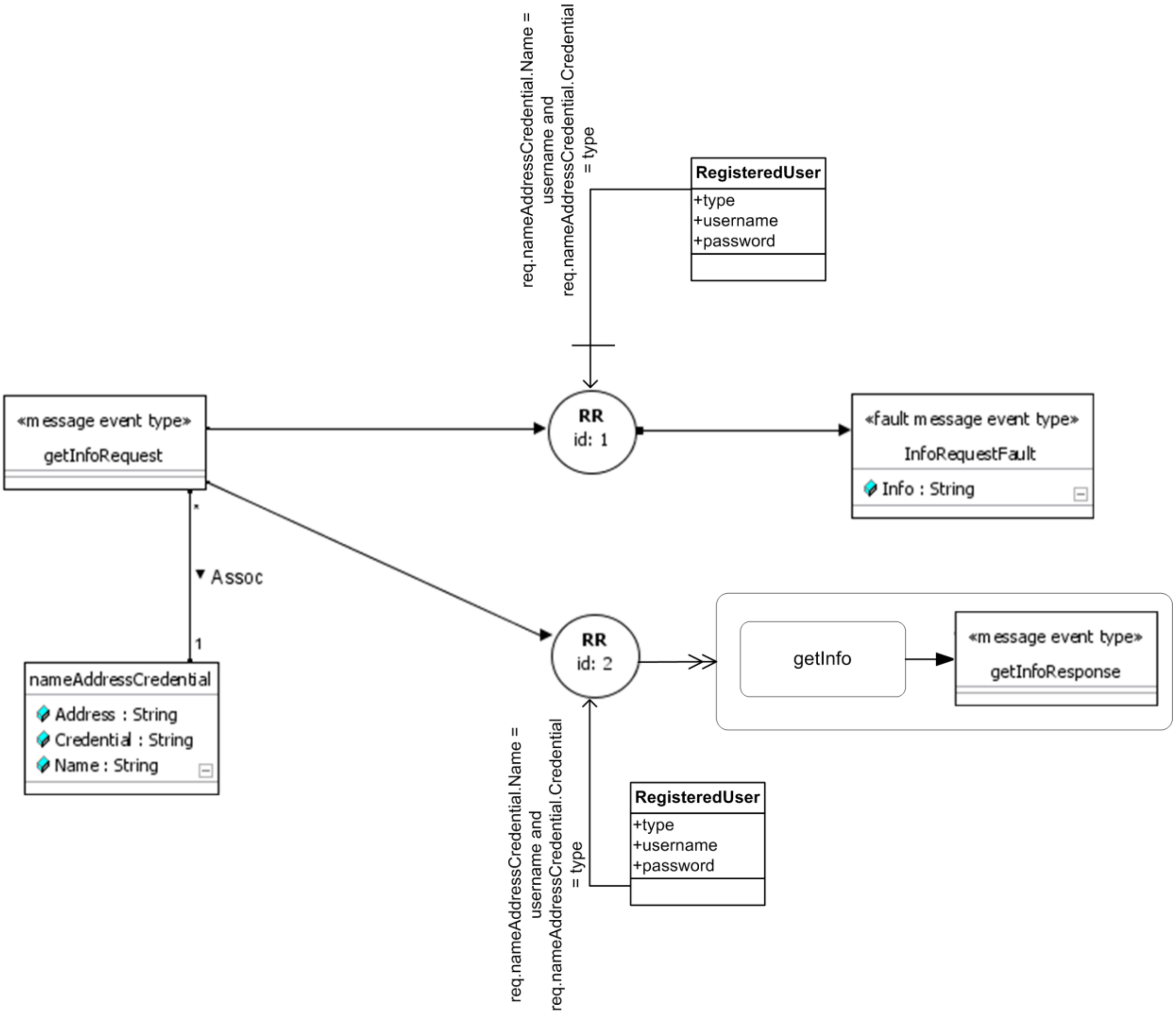


Figure 132. Associated reaction rules to the rule gateway from the rBPMN model of the In-Out pattern (with fault) shown in Figure 131

4.1.1.4. In-Optional-Out

This pattern consists of one or two messages: a message received by a service from some other node (i.e., input message), optionally followed by a message sent to the other node from the service (i.e., output message). Each message may trigger a fault in response as specified in the "Message Triggers Faults" model. In this MEP, the service could return a (fault or regular) message to its invoker as shown in Figure 133.

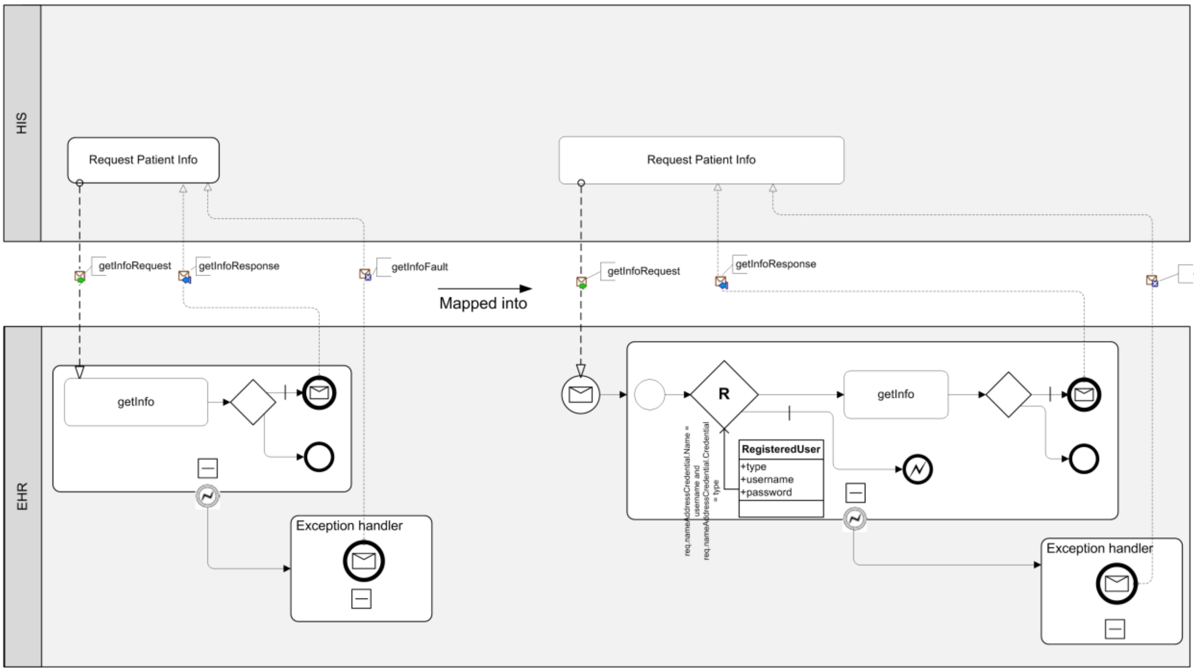


Figure 133. The In-Optional-Out pattern modeled in rBPMN

In the right part of Figure 133, we can see that the rule gateway could invoke the "getInfo" task or generate an error end event. Which of these sequences paths will be followed depends on the evaluated value of the condition of the rule gateway. After executing the "getInfo" task, a sub-process either ends or returns a (output) message to the service requester. This decision depends on the evaluation of the condition of second gateway. How to implement this optional decision (i.e., how can a service decide whether an optional message should be send or not) is not defined precisely in the specification of MEPs [137]. Because of this weak definition, we added a general data-based gateway after the "getInfo" task, so that a decision whether the message should be returned or not, can be taken based on data that could be changed in the "getInfo" task. This data gateway could be replaced with a rule gateway in order to capture even more advance business logic based on which the decision of returning the output message is sent.

In the case of this pattern, it is also important to notice that we provide the same support for exception handling, as we have already described on the In-Out pattern.

4.1.2. WSDL 2.0 Out-Bound MEPs

The out-bound patterns are patterns where a service first sends a message (i.e., output message goes first). In the following subsections, we present all the out-bound patterns. Their models in the standard BPMN are the same as models of the in-bound patterns with the main difference related to the direction of the message exchanged between different collaborating pools. In this subsection, we primarily focus on the description of the rBPMN models of outbound patterns. Given the nature of outbound patterns where the service initiates the interaction, there is no need to model services with reaction rules. That is, a reaction rule (i.e., service) is not triggered with an input message. In this case, the trigger for the service execution and for sending the output message is coming from the business logic captured on the side of the service. This might be a result of the change of the information state of the service side or some internal event. However, from the perspective of the service requester, this is not seen as reaction rule. Yet, we want to model message exchange patterns in a consistent way by using rBPMN rule gateways. In general, such rule gateway has a production rule associated, which indicates the condition under which the service is triggered, and consequently the output message sent.

This modeling approach also allows for generation of the complete service descriptions, similar to what we discussed with the inbound services.

We should note that outbound patterns are not so extensively discussed in the research literature comparing to the inbound patterns. There are very few (if any for all patterns) examples publicly available. There even is a lot of debate in the research community whether outbound patterns are useful or not. Moreover, there has been very limited research done on the development of modeling (or software development) languages that support this kind of patterns. To the best of our knowledge, WebML[7] is the only language that has an explicit support for this pattern. We decided to support these patterns, because they can be quite useful in publish/subscribe scenarios in large enterprises. In such situations, a service requestor will retrieve published services and consequently select services to use in its own systems. Moreover, the service requestor will be able to send parts of its business logic to the service provider side, so that the service provider can execute those services under conditions defined by the service requestor (e.g., the change of the stock prices) and notify the service requestor (via output messages) about the result of the service execution.

In the following subsections, we discuss in detail how we modeled each of the outbound patterns.

4.1.2.1. Out-Only

This pattern consists of exactly one message sent to some other node from a service. No fault may be generated. In Figure 134, we show the model of the Out-Only pattern in the standard BPMN and its mapped and refined version in rBPMN. This pattern is organized as follows. A service requester (e.g., EHR Locator) subscribes for a service (e.g., getInfo service of Police) sometime in the process before the message exchange of this pattern might happen. Along with its service subscription, the service requestor (e.g., EHR Locator) might have also sent its policies (rules) which define conditions under which the service (or message exchange) should be triggered. These conditions could be easily be updated or even extended by adding more derivation or production rules in a ruleset associated to the rule gateway (N.B., This will further be elaborated on in Section 4.4). The service provider (e.g., Police) received those rules and incorporated them in its business process, so that when those rules are fired, the service provider (e.g., Police) will inform the service requestor (e.g., EHR Locator) about requested information by sending the output message. This policy is represented with a rule gateway in the Police pool, under which the service should be executed (“getInfo”) and output message be sent (“getInfoRequest”). This can be defined through a production rule: *If a patient is supervised, then get the electronic health record and notify the* police. Then, the service requestor (EHR Locator) receives the message and based on its own business logic (e.g., defined by a rule gateway) defines what steps it needs to take (e.g., whether to perform the “Inform police” task or not).

[7] http://www.webml.org

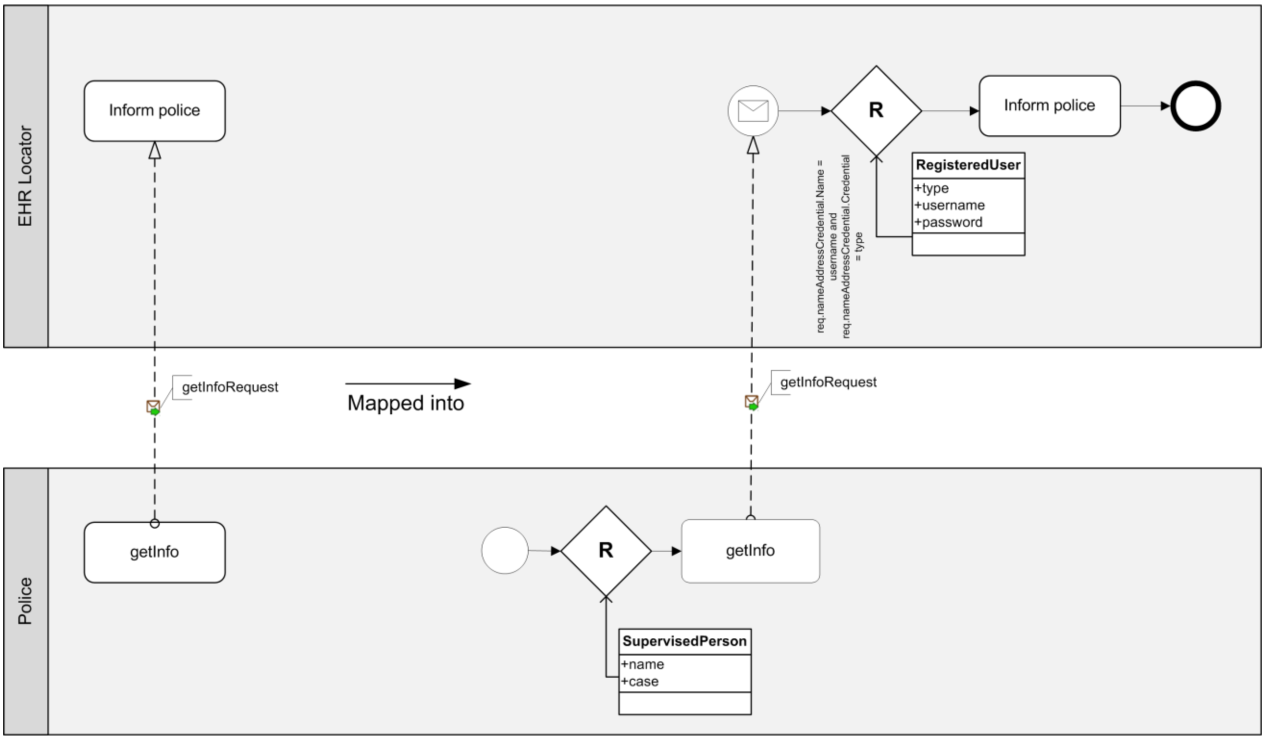


Figure 134. The model of the Out-Only pattern in rBPMN

In Figure 135, we show an example of the production rule that might be attached to the rule gateway on the service side (e.g., Police pool). This rules drives the decision on the service should it send the (output) message to the service requestor. This example also nicely explains a need for standard for rule interchange format (e.g., RIF). Our solution (rBPMN) is based on the integration of the rule interchange language (R2ML) and a process modeling language (BPMN). As such, it nicely demonstrates how those two standardization efforts can effectively be synergized.

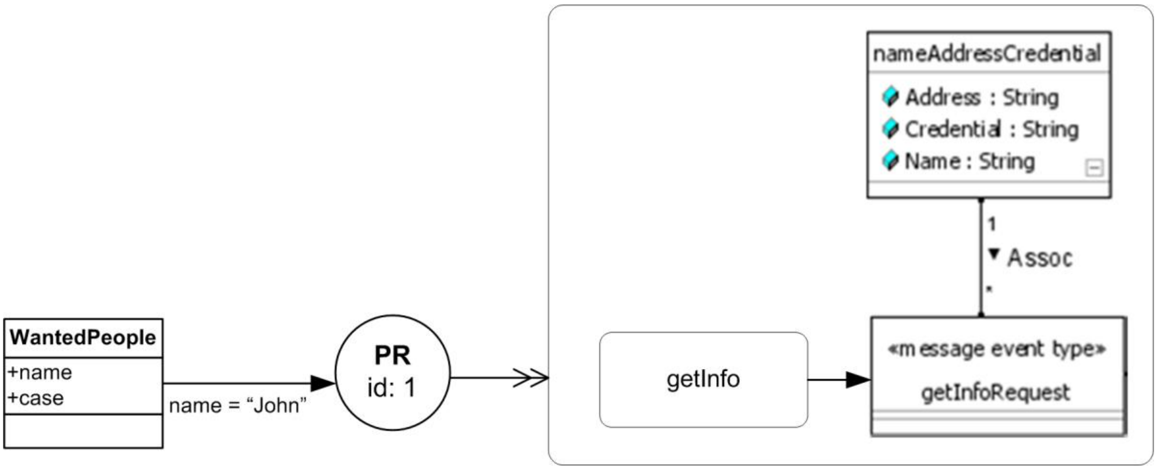


Figure 135. The production rule associated to the rule gateway (service side) from the rBPMN model of the Out-Only pattern shown in Figure 134

4.1.2.2. Robust Out-Only

This pattern can be considered a variation of the Out-Only pattern. It also consists of exactly one message sent to some other node from a service. The difference in this case is that faults can be triggered by the message as specified in the "Message Triggers Fault" model [137] and be sent to the initiator of the interaction.

In Figure 136, we the model of the Robust Out-Only pattern represented first in the standard BPMN and then in rBPMN. The Fault message handling is the same as for the Robust In-Only pattern, by using the subprocess called that Exception handler that we introduced to return the fault message to the sender of the initial message. In this case, the “getInfo” task sends a message to the service requestor (e.g., EHRLocator). The service requestor decides whether to return a fault message to the service or not based on its internal logic. This internal logic can be defined as through a rule gateway, which might be associated with a reaction rule. In the rBPMN model from Figure 136, we also show how the rule gateway can be used to distinguish between different types of faults those that are result of internal service execution errors and those that are generated when the condition of the rule gateway evaluates to false. Here, we should also mention that for both rule gateways used in model from Figure 136, we could associate different rules. We can associate production rules to the rule gateway on the service side, similarly as we showed for the Out-Only pattern. The rule gateway on the service side might be associated with a pair of reaction rules similar to those from Figure 132.

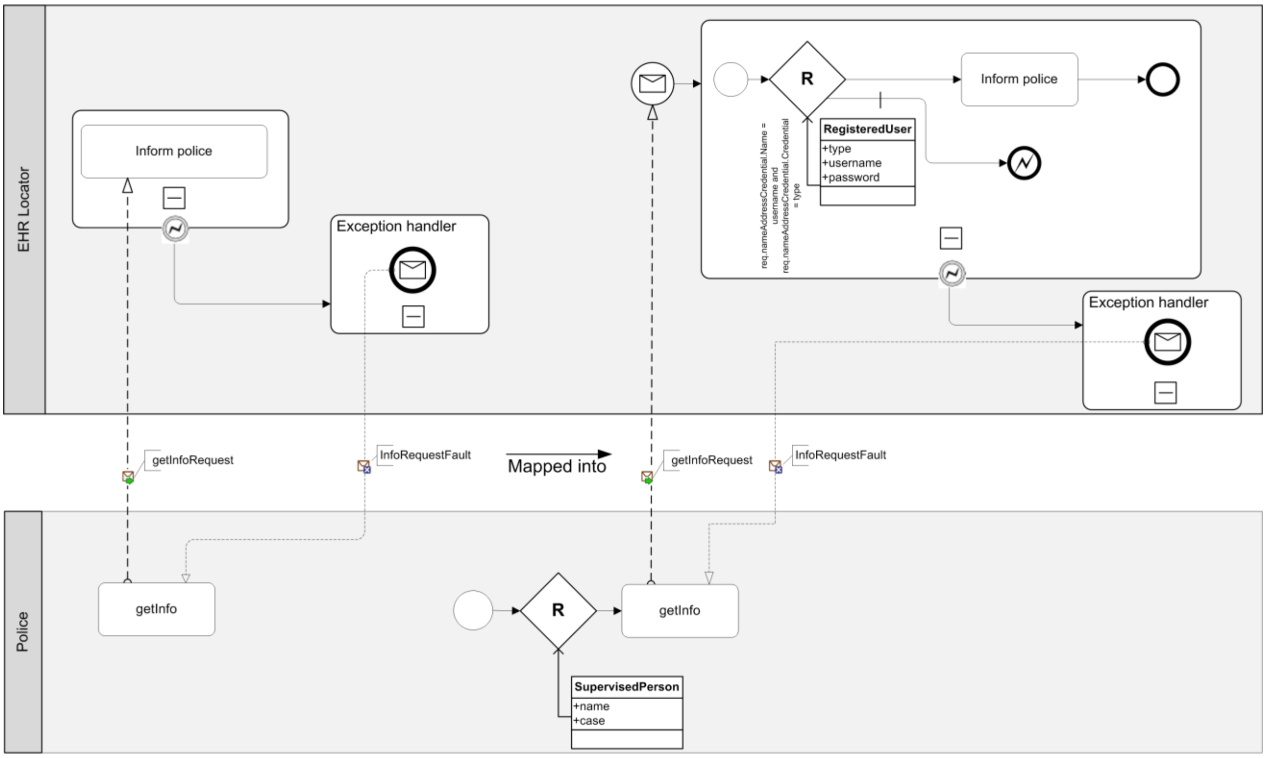


Figure 136. The model of the Robust Out-Only pattern in rBPMN

4.1.2.3. Out-In

This pattern consists of exactly two messages: a message sent to some other node from a service (i.e., output message), followed by a message received by the service from the other node (i.e., input message). The second message may be replaced by a fault as specified in the "Fault Replace Message" model [137]. In Figure 137, we show the Out-In WSDL pattern first modeled in the standard BPMN and then in rBPMN by using the notion of rule gateways. Here, the rule gateway is used to define a condition under which the service should be executed (e.g., the “getInfo” task) and output message be sent (e.g., “getInfoRequest”). The service requestor (HIS) receives the output message, which triggers its reaction rule associated with the rule gateway. Should the rule condition is satisfied, the business logic of the service requestor is performed (e.g., “Request Patient Info” task) and the input message (“getInfoResponse”) is forwarded to the service.

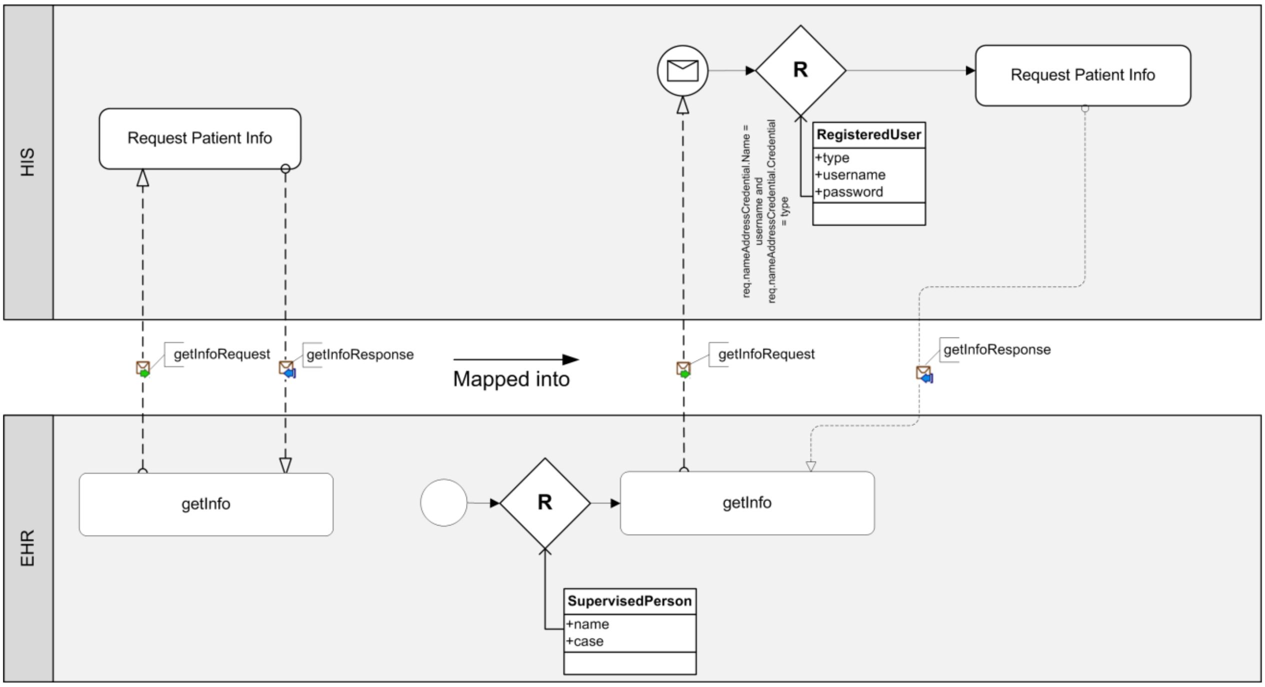


Figure 137. Representation of the Out-In MEP in the BPMN by using RR

As this pattern could return a fault message, we can support this through the rBPMN model given in Figure 138. This is similar to the In-Out pattern shown Figure 131, but now the fault message is sent from the service requestor to the service as per the standard definition of the Out-In pattern and the associated "Fault Replace Message" model [137]. Again, we used the Exception handler subprocess (but now on the service requestor side) to return the fault message to the initial message sender (the service). This exception may be generated by the rule gateway, depending on the evaluated value of its condition, which is actually used to decide should the “Inform police” task be invoked or not, or during the execution of this task. Similar types of rules can be associated to the rule gateways as we indicated for the Robust Out-Only. The only difference from the Out-Only pattern is the reaction rule that is fired when the condition is evaluated to true (e.g., Info Police task in our example from Figure 138). In the Out-In case, the triggered action/event part of the reaction rule is a sequence of the task executing and input message (e.g., “getInfoResponse”), while Out-Only does not have the input message.

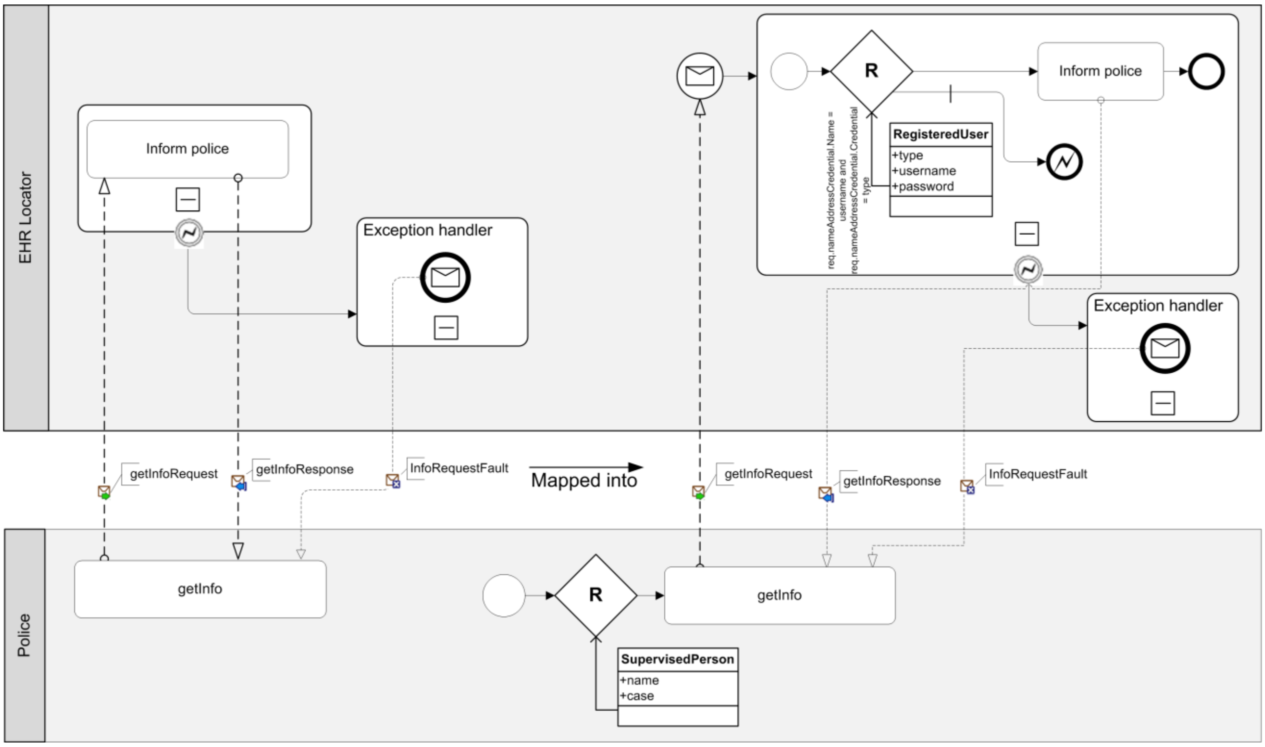


Figure 138. Model of the Out-In (with fault) pattern in rBPMN

4.1.2.4. Out-Optional-In

This pattern consists of one or two messages: a message sent to some other node from a service, optionally followed by a message received by the service from the other node. Each message may trigger a fault in response as specified in the "Message Triggers Faults" model [137]. We model this pattern (with fault handling) in rBPMN as shown in Figure 139.

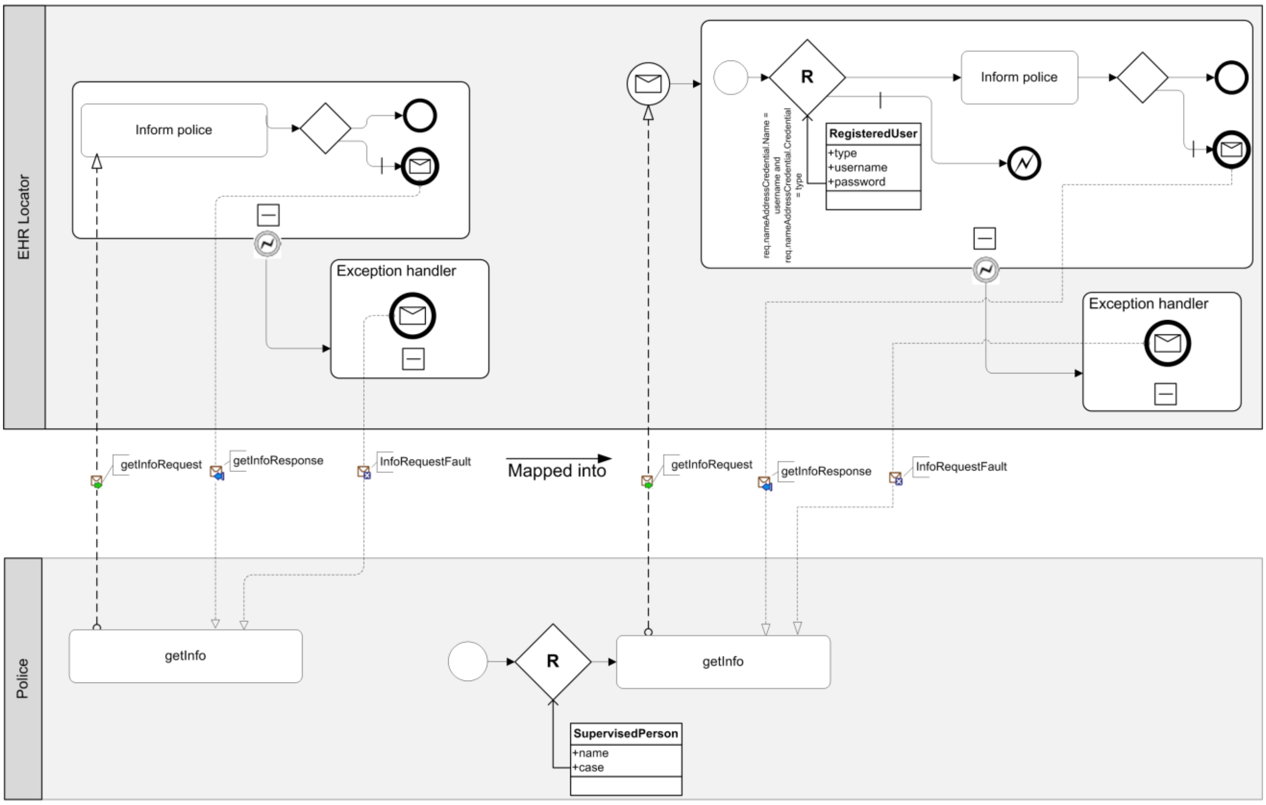


Figure 139. Model of the Out-Optional-In pattern in rBPMN

This pattern is similar to the In-Optional-Out pattern, but it has the opposite direction. In Figure 139, we can see that the service requestor could whether to return an input message or not after completing its internal logic (modeled with the “Inform police” task in Figure 139). Yet, given that this pattern is associated with the "Message Triggers Faults" model, the rule gateway on the service requestor side can decide generate a fault message and send it to the service via the Exception handler. Additionally, the fault message can be also sent to the service if an error occurs during execution of the “Inform police” task. Finally, we should mention that we could associate rule to the all rule gateways shown in Figure 139 by following the similar rule association principles described in the previous pattern (Out-In with fault). The only difference is the data gateway added after the “Inform Police” task in Figure 139. This way, the decision whether a message should be returned or not, is based on the data changed during the “Inform Police” task execution. This data gateway might be replaced with a rule gateway, which might have an associated rule or set of rules (similar to the In-Optional-Out pattern).

4.2. Interaction modeling in rBPMN

The workflow patterns (see Section 4.3) describe a control flow of a business process. A control flow is a perspective that characterizes process orchestrations. However, there are several differences between process orchestrations and process choreographies that need specific consideration: choreographies are based on message exchange, and potentially many participants interact in a choreography, while orchestrations are based on a control flow between the activities of a single process performed by a single organization.

Service interaction patterns (SIP) aim at describing a set of small granular types of interactions that can be combined in choreographies [1], and they are independent of concrete languages or implementations. Similar to workflow patterns (see Section 4.3), service interaction patterns cannot be modeled with the standard BPMN, as BPMN lacks some important concepts for representing choreographies [22], so we proposed those extensions in rBPMN in order to represent choreographies as shown in Section 3.1.4. Through the service interaction patterns represented in subsequent section, we will show implications of rBPMN extensions for modeling choreographies.

There are two approaches to modeling of choreographies: interaction models and interconnected interface behavior models (interconnection models) [22]. Interaction models are built up of basic interactions (message exchanges), while interconnected interface behavior models defines control flow of the each participant in interaction (choreography). The representatives for the interaction model approach are WS-CDL [55], Let's Dance [150] and iBPMN [25]. Interconnected interface behavior models can be represented in BPMN [ref] and BPEL4Chor [ref]. As BPMN can be used for both modeling approaches, we here show basic interaction patterns expressed in rBPMN as both interaction and interconnected interface behavior models.

We should note that interaction models do not suffer from two drawbacks of interconnected behavioral interface models: *redundancy* (parallelism, branching and loops are duplicated in the model, as each collaborating party needs to represent business flow from their own perspective) and *potentially incompatible behavior* (potential deadlocks because of sequencing structures, which do not mach properly) [22]. However, it is has been reported that interaction models suffer from their anomalies too, such as locally unenforceable choreographies [22].

Authors in [1] recognized 13 service interaction patterns, which are divided in to four groups, following the three main characteristics: i) number of parties involved (i.e., bilateral and multilateral interactions); ii) maximum number of messages exchanged in an interaction (i.e., single or multi-transmission interactions) and iii) in the case of two-way interactions, whether the receiver of the response is in the same time as the sender of request (round-trip and routed interactions). We will present these patterns as described in [6].

4.2.1. Single-transmission Bilateral Interaction Patterns

There are three patterns in this group. This group of patterns include basic send/receive patterns where one participant send a message and wait for a response.

4.2.1.1. Send

The send pattern represents a one-way interaction between two participants seen from the perspective of the sender. There are different flavors of this pattern, considering, for instance, the scenario when the sender selects the receiver, that is, the receiver is known either at design time of the choreography or only during the execution of a conversation.

Figure 140 illustrates an example where one participant (represented by a BPMN Pool 1) is sending a message to another participant (Pool 2). This is represented as an interconnected behavioral interface, which is a view of individual partners onto the choreography. We represented partners in message exchanges as BPMN pools. The sending of the message is done by the Send task in Pool 1, after rule gateways' R_1 condition evaluates to true, while the receiving of this message is done by using receiving message event in Pool 2, which receives the message. After receiving the message the sequence flow goes to the rule gateway R_2 that continues sequence flow if the reaction rule's condition (Condition1) evaluates to true. We employ production rule, which is attached to the rule gateway R_1 on the service side (e.g., Pool 1). This rule drives the decision on the service should it send the (output) message to the service requestor.

This pattern can be described in plain BPMN, however by using rules we gain important advantages, such as that a service requester (Pool 2) could subscribe to a service (e.g., Send task of Pool 1) sometime in the process before the message exchange of this pattern might happen. Along with its service subscription, the service requestor (e.g., Pool 2) might have also sent its policies (rules) which define conditions under which the service (or message exchange) should be triggered. These conditions could be easily being updated or even extended by adding more derivation or production rules in a ruleset associated to the rule gateway. The service provider (e.g., Pool 1) received those rules and incorporated them in its business process, so that when those rules are fired, the service provider (e.g., Pool 1) will inform the service requestor (e.g., Pool 2) about requested information by sending the output message.

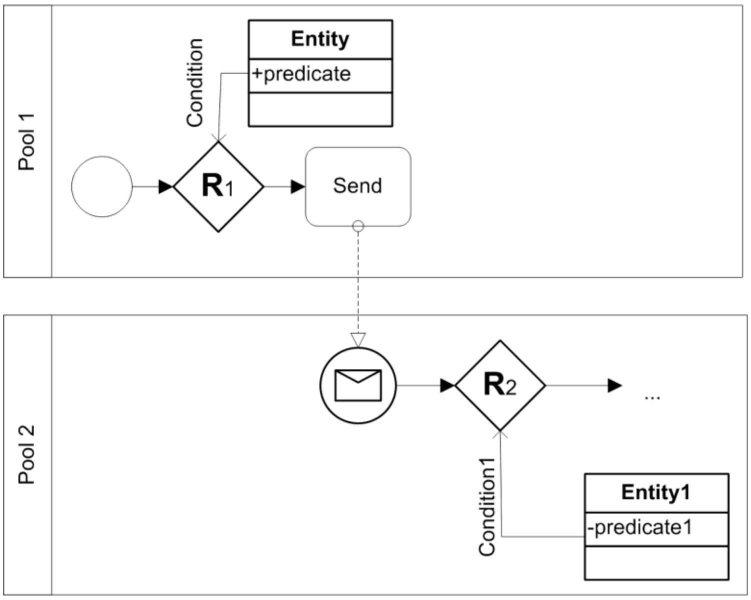


Figure 140. The "Send" pattern

In Figure 141, we show the interaction model for the "Send" pattern where only relevant interactions and dependencies between nodes are shown. The interaction model includes only pools and message flows between pools, as well as the gateways. The interaction model shown in Figure 141 is attained as a refinement of the model shown in Figure 140 in a following way: for every message flow, send and receive events are introduced. In this model, we show the message flow between Pool 1 and Pool 2, annotated with the start message event. This event represents a message send from Pool 1 to Pool 2. After sending this message, the sequence flow goes to the rule gateway R_2 that decides based on its condition whether the sequence flow will continue or not. The rule gateway is associated to one node in the scenario. It this case, the association defines who actually carries out the choice. In the case of the model from Figure 141 that is Pool 2. We should note that rule gateway R_1 is not shown in this interaction model, as we consider it as a part of Pools' 1 internal logic.

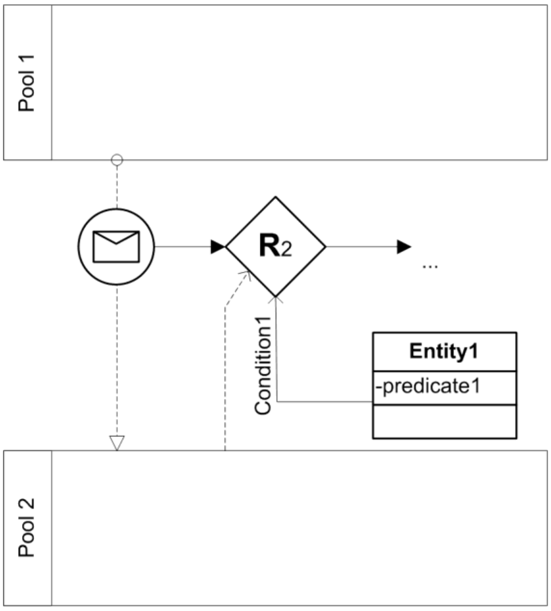


Figure 141. The “Send“ pattern (interaction model)

This rBPMN model of the Send service interaction pattern can directly mapped into the Web Service Out-Only and Robust Out-only message exchange patterns (MEP), because receiving participant can return a fault message, as we have shown in Section 4.1.1.1.

4.2.1.2. Receive

The receive pattern also describes a one-way interaction between two participants, but this time seen from the perspective of the receiver. In terms of the message buffering behavior of the receiver, two cases can be distinguished. Messages that are not expected are either discarded or stored until a later point in time, when they can be consumed.

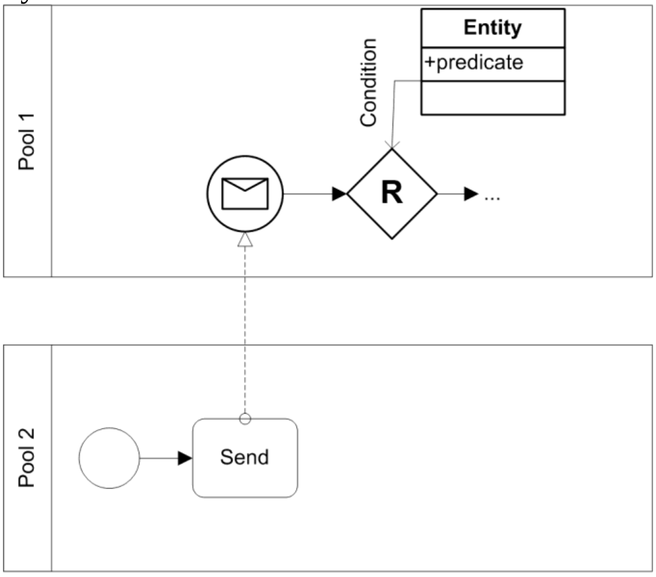


Figure 142. The “Receive” pattern

We show an interconnected behavioral interface model of the receive pattern in Figure 142. The receipt of the message is represented by receiving a start message and a rule gateway. If a reaction rule attached to the rule gateway evaluates to true sequence flow is continued. This establishes an explicit tractability between the rule gateway and the rule’s full definition represented in R2ML. This enables to further define the rule with (post-)conditions and connect the service definition with the vocabulary elements.

A corresponding interaction model for the receive pattern is shown in Figure 143. The mapping is similar to one described for the “Send” pattern (see Section 4.2.1.1).

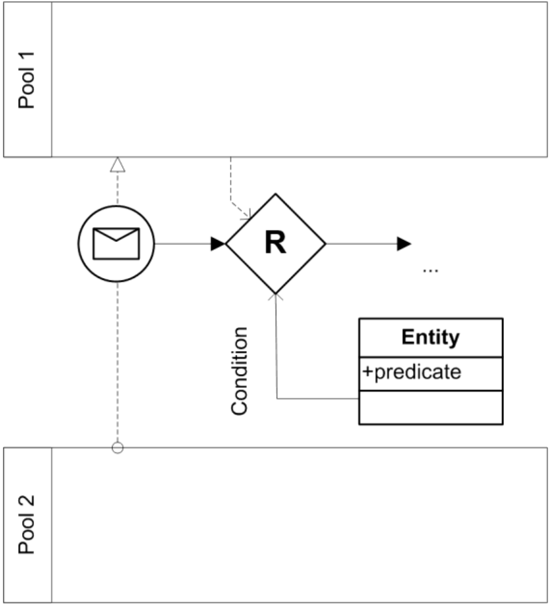


Figure 143. The “Receive“ pattern (interaction model)

This pattern is supported in WSDL with two MEPs: In-Only and Robust In-Only, as shown in Section 4.1.1. Due to its stateless nature, WSDL cannot define under which conditions the message should be accepted or discarded, so for defining such precondition we propose the usage of the rBPMN rule gateway. For example, if a message arrives earlier than that message can be accepted.

In this pattern, we used a reaction rule attached to rule gateway because reaction rule has an event for its input, so when a message event happens it fires the reaction rule. In this case, a production rule cannot be used, as production rules are not triggered on an event, but rather on a true condition. The use of reaction rules is convenient because we can model an input message and output service activity. By using the rule gateway in this pattern, we add additional elements of the business logic, which can be used to define a condition under which the modeled service (task) can be used, once the input message has been received. Most importantly, this condition can be updated both at run- and design-time. For example, this condition can define from which partner service our modeled service can process requests.

4.2.1.3. Send/Receive

In the send/receive pattern, a participant sends a request to another participant who then returns a response message. Both messages belong to the same conversation. Since there could be several send/receive interaction instances happening in parallel, corresponding requests and responses need to be correlated. This pattern represented in rBPMN is shown in Figure 144. The message is send from Pool 1 to Pool 2, where this message is received with a message event, followed by a reaction rule attached to the rule gateway. Based on the condition defined on this rule, the Send task is invoked and the message is returned to Pool 1. By employing a reaction rule attached to the rule gateway here, we could precisely define a condition on which a message could be returned from the Pool 2. We used reaction rule, because we have an event for its input and a task for its triggered action.

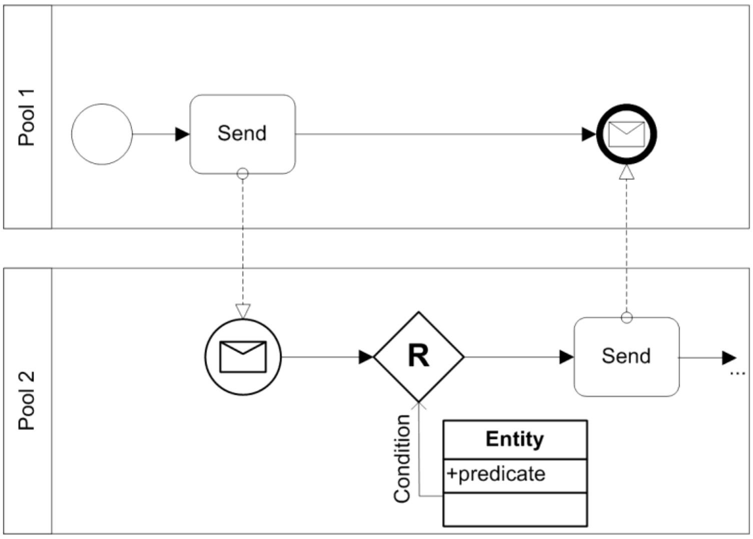


Figure 144. The “Send/Receive” pattern

The “Send/Receive” pattern from Figure 144 represented as an interaction model in rBPMN is shown in Figure 145. We followed the same principle of mapping as for the “Send” and “Receive” patterns, where message flows are annotated with message events and a rule gateway is associated with a pool that is responsible for its execution (i.e., Pool 2 in Figure 145).

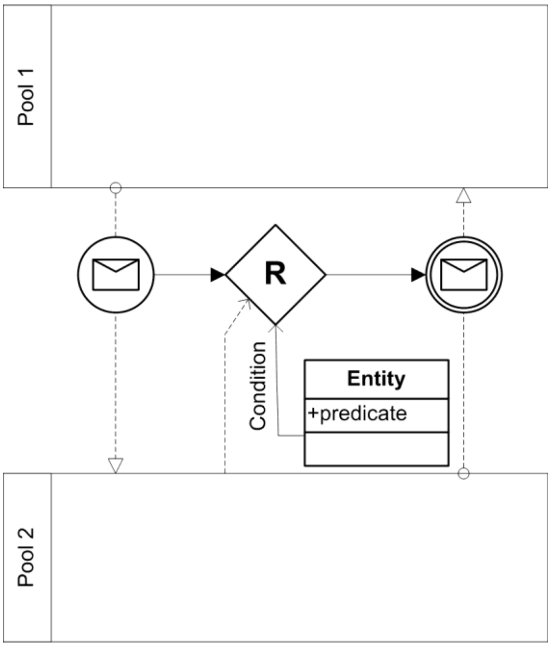


Figure 145. The “Send/Receive” pattern (interaction model)

This pattern has two corresponding WSDL MEPs, Out-In and Out-Optional-In as shown in Section 4.1.2. As both interactions in this pattern may result in a fault message in response, the same exception handling process can be used as described for Out-Optional-In MEP (see Section 4.1.2.4).

4.2.2. Single-Transmission Multilateral Interaction Patterns

In this group, authors of [6] identified four patterns that deal with multilateral interactions. In this group of patterns, one participant may send or receive multiple messages where these messages are part of different interactions threads that belongs to different participants (i.e., one to many and many to one interactions).

4.2.2.1. Racing Incoming Messages

The racing incoming messages pattern is described as follows: a participant is waiting for a message to arrive, but other participants have a chance to send a message. These messages by different participants "race" with each other. Only the first message arriving will be processed. The type of the message sent or the category, to which the sending participant belongs, can be used to determine how the receiver processes the message. The remaining messages may be discarded or kept for later consumption. We model this aspect of message racing in this pattern by using an event-based XOR gateway and a rule gateway as shown in Figure 146. The figure contains the interconnected behavioral interface model of the pattern. In this case we use multiple tasks from a Pool 1 to send multiple messages, but also, multiple pools that would send a message could be also used.

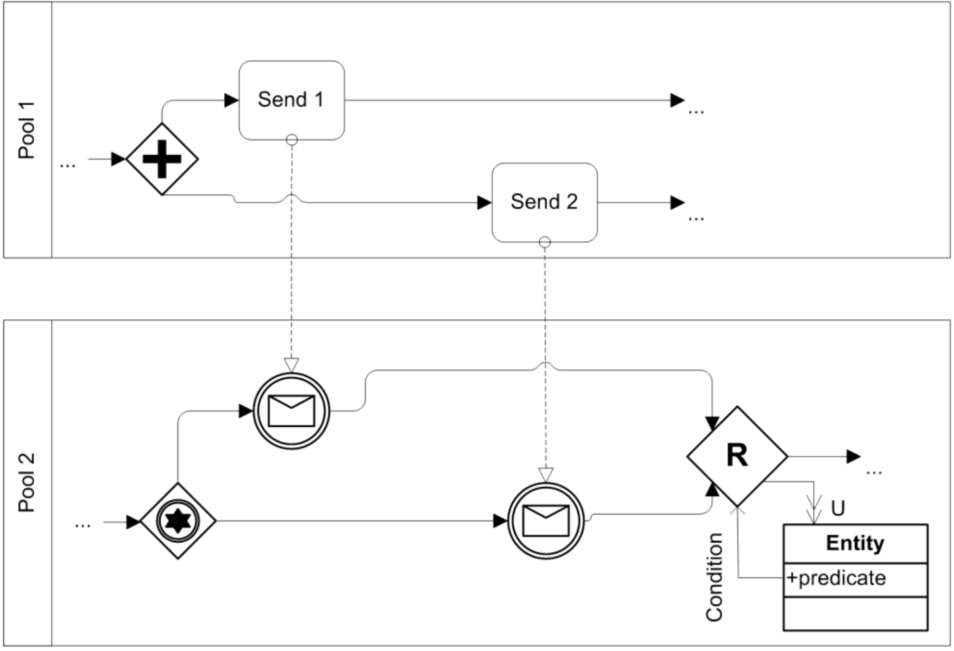


Figure 146. The "Racing incoming messages" pattern

Figure 146 shows a scenario where Pool 2 has done some activities and now waits for the Pool 1 message. If the Send 1 task sends the message, the intermediate message event in Pool 2 receives the message and then the reaction rule attached to the rule gateway is fired and continues a sequence flow if its condition is satisfied. At the same time, when this happens, the reaction rule attached to the rule gateway updates the Entity's predicate used for its condition, so that any other message that arrives after this first message will not be consumed anymore. When this happens, if now the Send 2 task sends the message, the rule gateway will hold execution of this sequence flow, that is, no further message from the racing participants. This is the case where a reaction rule attached to the rule gateway can be used in this pattern. We used reaction rule as we have event for an input, and also a produced action (Entitys' update), which could be modeled by a reaction rule. Without the rule gateway, it would be hard to implement this pattern in plain BPMN, because it is not possible to define such an (runtime-updatable) condition on a rule.

The "Racing incoming messages" pattern represented as an interaction model in rBPMN is shown in Figure 147. In this interaction model, we have shown just interactions between the partners (pools) involved in this scenario. Every message event represented as interaction is attached to a message flow. Rule gateway is connected to Pool 2 with dashed line, who carries out the choice. The decision logic is the same as in Figure 146. In interaction model (see Figure 147) in difference to corresponding interconnection model (see Figure 146) we can only see interactions between participants, but not their internal logic, which is an important aspect in modeling service choreographies.

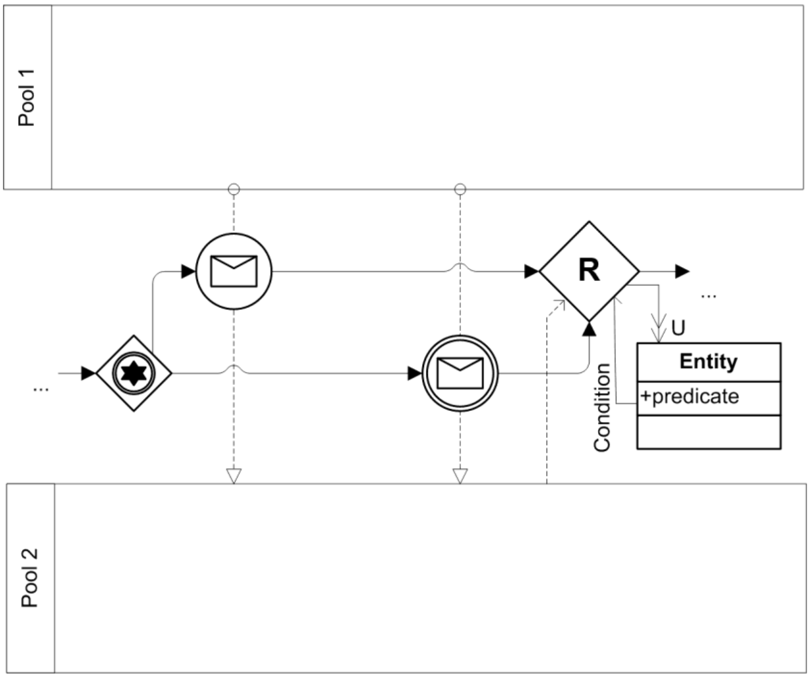


Figure 147. The “Racing incoming messages“ pattern (interaction model)

4.2.2.2. One to Many Send

A participant sends out several messages to other participants in parallel. It might be the case that the list of recipients is already known at design-time of the choreography or, alternatively, the selection of the recipients takes place in the course of the conversation (i.e., at run-time).

In Figure 148, we give an rBPMN (interconnected behavioral interface) model of this pattern by using a multiple instances task (“Send 1”) that sends messages to each participant contained in the participant set (of type Pool 2). Participant set represents a set of participants of the same type, used in the same conversation. We represent participant set by an rBPMN artifact with <ref> name. We assume that all referenced participants are of the same type. The introduction of the participant set, which is used to determine to whom the message is sent, is needed as regular BPMN cannot capture it. This pattern is similar to the Send pattern, but in this case one participant in conversation is sending messages to multiple participants. When Pool 2 receives the message, it fires a rule gateway (with an associated reaction rule), which evaluates if the sequence flow should be continued based on the rule gateway’s condition and potentially invoke some task or not. The multiplicity of the Pool 2 is denoted with a small parallel indicator (|||) displayed at the bottom-center of the pool. This means that we can have multiple instances of a Pool 2. References to those instances are contained in a participant set.

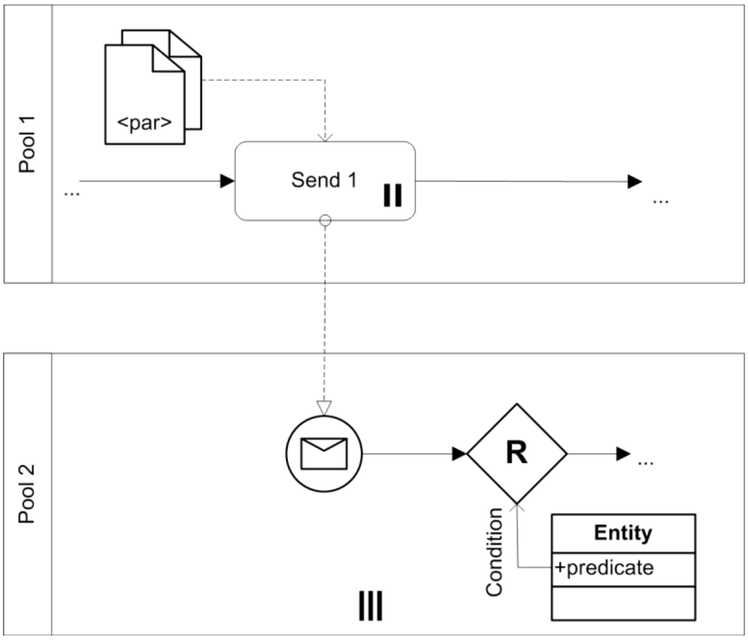


Figure 148. The “One-to-Many Send” pattern

The “One-to-Many Send” pattern represented as an interaction model in rBPMN is shown in Figure 149. In this model, we have a message being sent through a multiple instance subprocess (i.e., Send) that is mapped from the multiple instance sending task used in the interconnected behavioral interface model given in Figure 148. The send message event is generated for each participant contained in participant set, attached to this message event. When the receiving side (Pool 2) receives the message, it uses the rule gateway to decide whether the sequence flow should continue or not. Of course, here, in the case of the else branch on the rule gateway, we can have a decision on what direct the sequence follow will take.

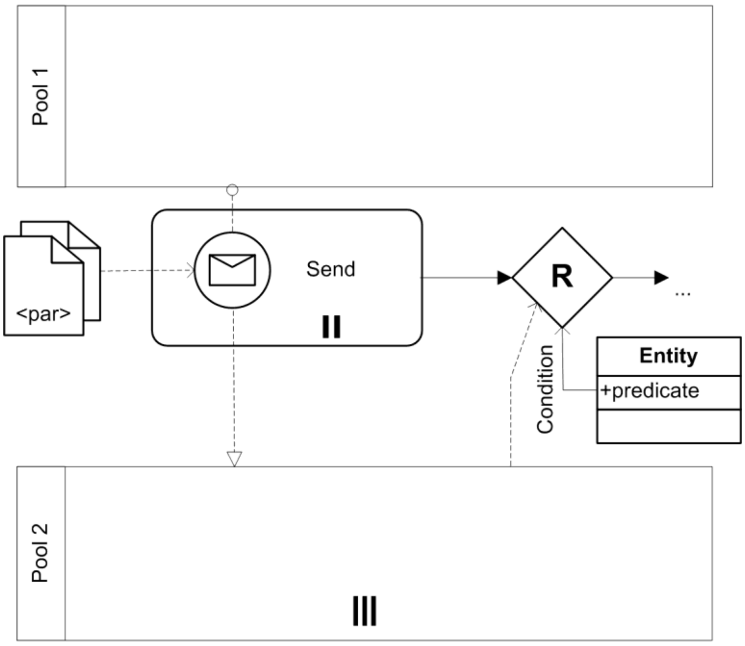


Figure 149. The “One-to-Many Send“ pattern (interaction model)

4.2.2.3. One from Many Receive

In the one-from-many receive pattern, messages can be received from many participants. In particular, one participant waits for messages to arrive from other participants, and each of these participants can send exactly one message. Typically, the receiver does not know the number of messages that will arrive, and stops waiting as soon as a certain number of messages have arrived or a time-out has occurred. An rBPMN interconnected behavioral interface model of this pattern is shown in Figure 150. The references to the senders are collected in a participant set created in Pool 2. An important aspect of this pattern is a notion of *stop condition* [6], which denotes a completion of the interaction. This condition could be expressed by a predicate over the messages received. Thus, we use a reaction rule attached to the rule gateway to define this condition. When a message arrives from the sender, the reference to the sender is stored in the participant set, and then the reaction rule attached to the rule gateway (R_1) is evaluated to decide if the stop condition evaluates to true (in this case when participant set size is equal to some given size). In this case, the interaction is considered complete. Otherwise, the sequence flow is returned to the event-based gateway to wait for a next message. In this pattern, it is important that an interaction occurs in a given period of time, because this determines whether the interaction has been successful or not when timeout. This is called a *success condition*. For this condition, we introduced an intermediate timer event that fires when a timeout happens, and then the reaction rule attached to the rule gateway (R_2) is invoked to decide if a message interaction is successful or failed, based on messages received. In the interaction was not successful, the reaction rule attached to the rule gateway R_2 evaluates to true, and the triggering of this rule gateway R_2will generate an exception. If reaction rule attached to the rule gateway R_2 evaluates to false, and sequence flow is returned to the wait for a next message. Without the rule gateways, it would be hard to define these conditions in plain BPMN, and impossible to change them in runtime. Rules enable to dynamically change the process conditions (and flows) by using a rule gateway, because of declaratively nature of rules.

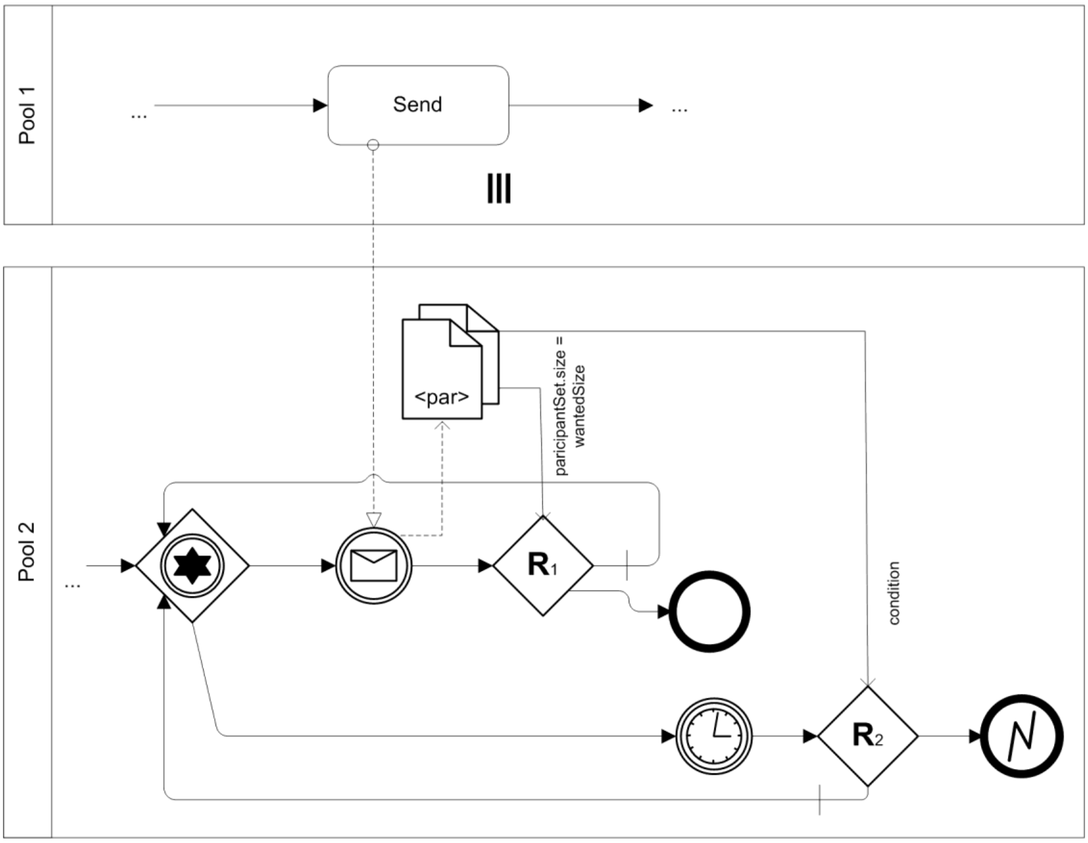


Figure 150. The “One from Many Receive” pattern

The “One-from-Many Receive” pattern represented as an interaction model in rBPMN is shown in Figure 151. In this model, we introduced repeating subprocess that contains an intermediate message event, which receives the message from the participant (Pool 1). When the message is received, the reference to the participant who has sent the message is stored in the participant set, and the sequence flow continues to the rule gateway (R_1). The rule gateway then decides based on the predefined condition (participantSet.size = wantedSize) whether this interaction should be completed or not (*stop condition*). When a timeout occurs, the repeating subprocess ends, and then the outside rule gateway (R_2) based on its condition decide whether sequence flow is going to return to this subprocess again for new set of interactions or to raise an exception (*success condition*). In interaction model, we shown all interactions and process control logic between the participants (pools), which enables to hide internal participant logic.

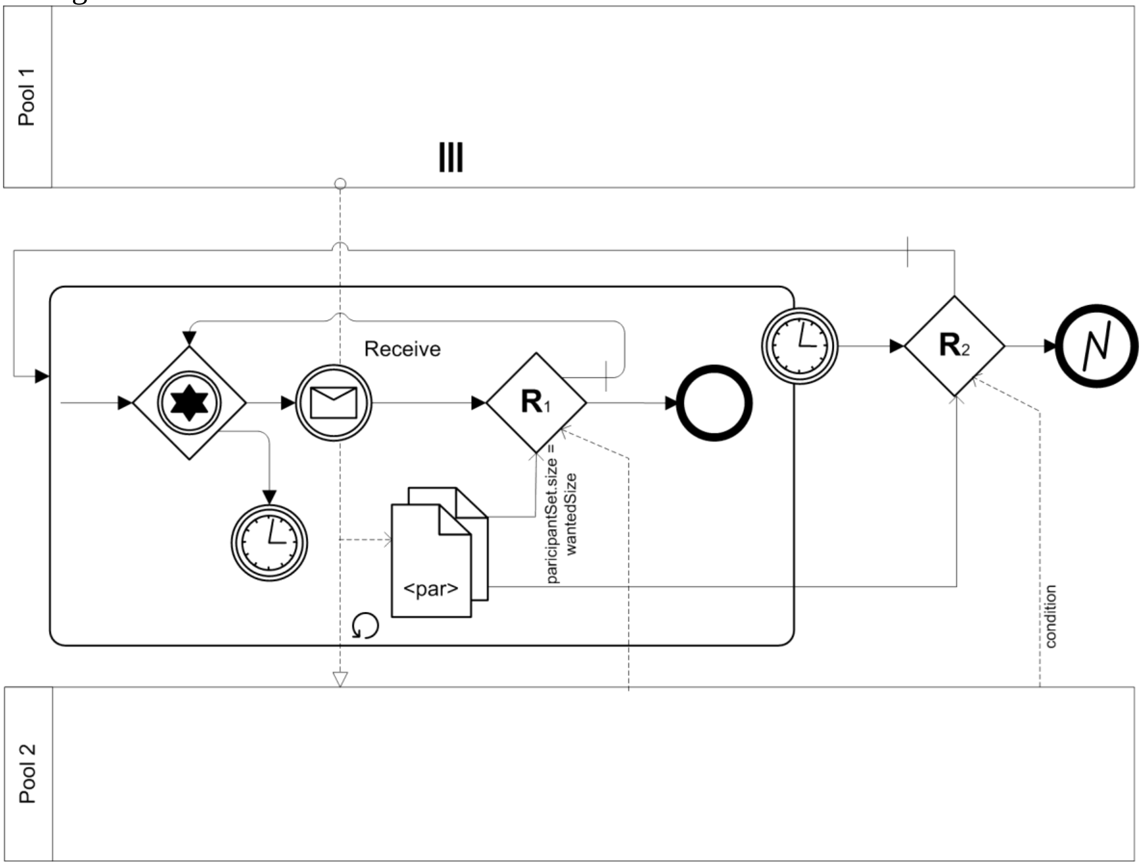


Figure 151. The „One-from-Many Receive“ pattern (interaction model)

4.2.2.4. One to Many Send/Receive

In this pattern, a participant sends out several requests to other different participants and waits for their responses. Typically, not all responses need to be waited for. The requester rather waits for a certain amount of time or stops waiting as soon as enough responses have arrived (e.g., given number of messages). An rBPMN interconnected behavioral interface model of this pattern is shown in Figure 152. The correlation between send/receive messages is done by pointing to the participants that should be included from the associated participant set. In Figure 152, we can see that a multiple-instance subprocess with the “Send” task is used to send messages to the other partners (Pool 2), by using information about partners from the participant set (<par>). The reason for such a design decision is because a number of partners may or may not be known at design time. When such a message gets to the partner (Pool 2), this partner uses its own logics represented with rule gateway (R_2) to decide whether it should return a message to the sender or not by invoking its own “Send” task. When a response from a partner (Pool 2) is received, the reference to the partner who has sent the message is stored in the participant set (<par1>), and a reaction rule attached to the rule gateway (R_1) is invoked to evaluate if the requested number of messages is received (i.e.,. this is the *stop condition*). It is possible that no response is received. During this process a timeout can occur (supported by an intermediate message event on the subprocess edge), and in this case, we use another reaction rule attached to the rule gateway (R_3) to decide if the *success condition* is achieved or not. If not, an end exception event happens. If yes, the sequence flow goes to the start to wait for new set of interactions.

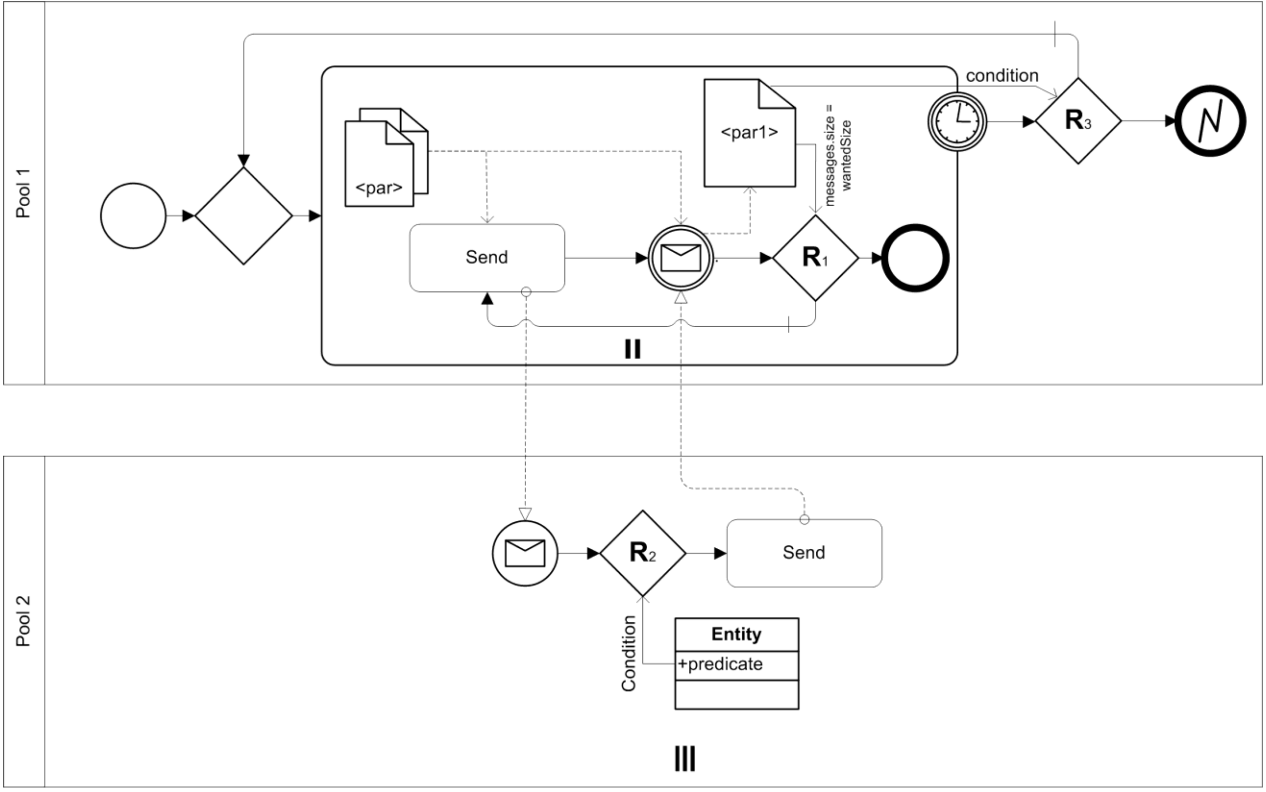


Figure 152. The “One to Many Send/Receive” pattern

The “One-to-Many Send/Receive” pattern represented as an interaction model in rBPMN is shown in Figure 153. In this model, we introduced a repeating Subprocess in which we have Send and Receive message flows annotated with message events. Messages send and received by those message flows, are send and received by participants referenced by the participant set (<par>). After the “Send” message from Pool 1 to Pool 2, a reaction rule attached to the rule gateway (R_2) is used to decide whether the response from Pool 2 should be received. When the message is received from Pool 2, another reaction rule attached to the rule gateway (R_1) is used to evaluate whether the *stop condition* is satisfied (i.e., wanted number of messages is received), and if so the interaction completes. During this conversion, a timeout may occur (by using an intermediate timer event on the Subprocess edge) and then the third rule gateway (R_3) is used to evaluate whether the *success condition* is achieved, and in that case exception occurs. If not, sequence flow is returned to the start to wait for a new set of interactions.

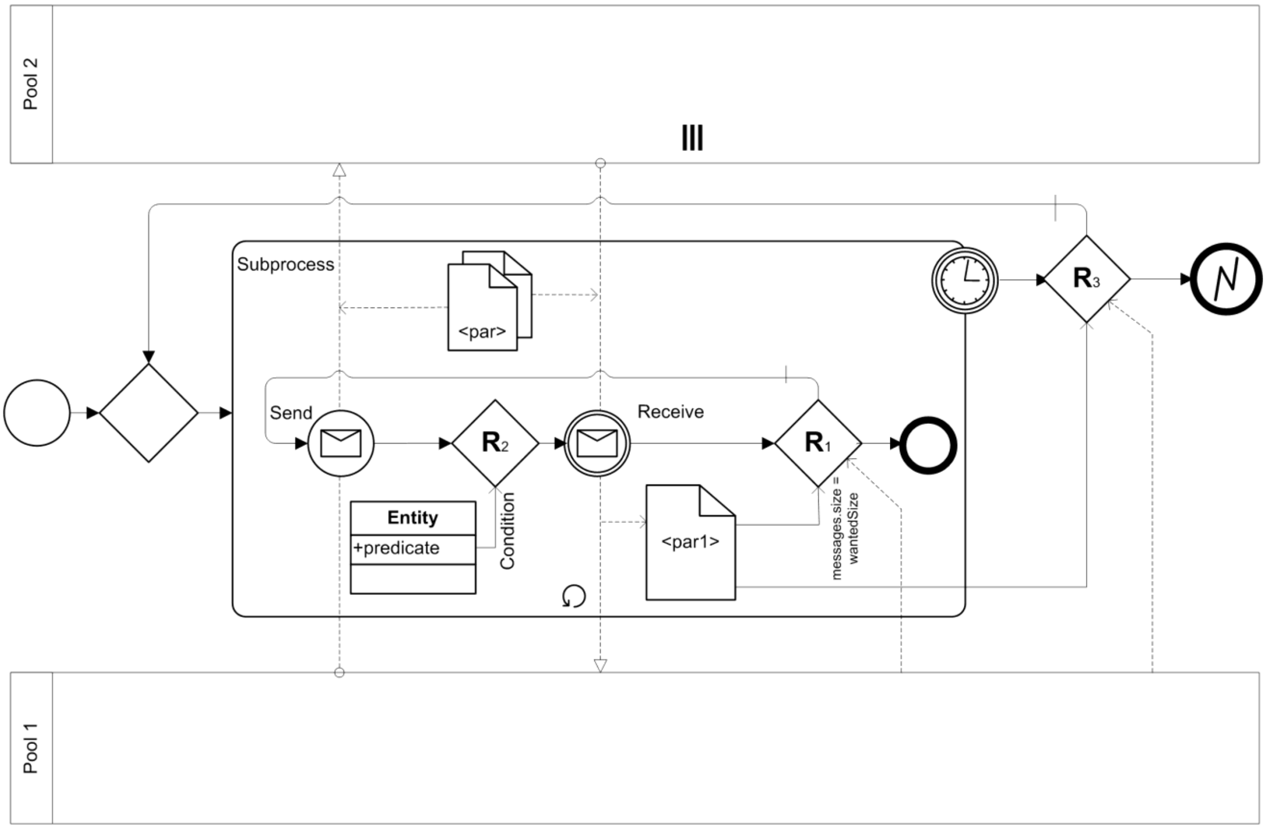


Figure 153. The “One to Many Send/Receive” pattern (interaction model)

4.2.3. Multi-transmission interaction patterns

The multi-transmission interaction patterns are the patterns where one participant sends (receives) more than one message to (from) the same logical participant. In this group, there are three patterns as described in the following three subsections.

4.2.3.1. Multi-responses

In the multiple responses pattern, a participant sends a request to another participant who sends back multiple messages. An important question in this scenario is how the requester knows that there are no more messages to be expected. One option would be that the messages contain information about whether there will be more messages or not. Another option could be that the last message is of a special type. Finally, also a time-out or a rule condition could be used to stop waiting for further messages. This scenario is shown in Figure 154. The participant (Pool 1) sends a request to the other participant (Pool 2) in the process, and subsequently Pool 1 receives a number of messages from Pool 2. Pool 2 uses its own logic and loop interactions to determine whether it should send the messages to Pool 1. This logic of Pool 2 can be represented by using a reaction rule attached to the rule gateway such as rule gateway R_2 in Figure 154. Pool 1 receives messages from Pool 2 and then Pool 1’s sequence flow goes to the “Task”, which is preformed, and then to a reaction rule attached to the rule gateway (R_1) to evaluate the *stop condition* for this process. If a reaction rule attached to the rule gateway R_1 evaluates to true, the process ends and the rule gateway updates the Entity’s predicate in order to stop the process if Pool 2 continues to send the messages after the stop condition. If the reaction rule attached to the rule gateway R_1 evaluates to false, the sequence flow goes to the event-based gateway to wait for another event. Besides the stop condition, a timeout can occur and this is represented with intermediate timer event in Pool 1. In addition, Pool 2 can determine based on its own logic when the multi-transmission should stop, and then Pool 2 returns the end message to Pool 1. This means that the stop condition is implemented on the Pool 2 side. In that case, Pool 1 receives the message and the sequence flow goes to the end message event, as for two other cases (timeout and rule-based stop condition). Then the end message event from the Pool 1 sends the message to the sending task in Pool 2 to inform it that no further messages will be sent any more.

In plain BPMN, it is hard to precisely define abovementioned condition on both participant sides. This means that those conditions are not usually represented in BPMN, and so this pattern cannot be fully supported in BPMN.

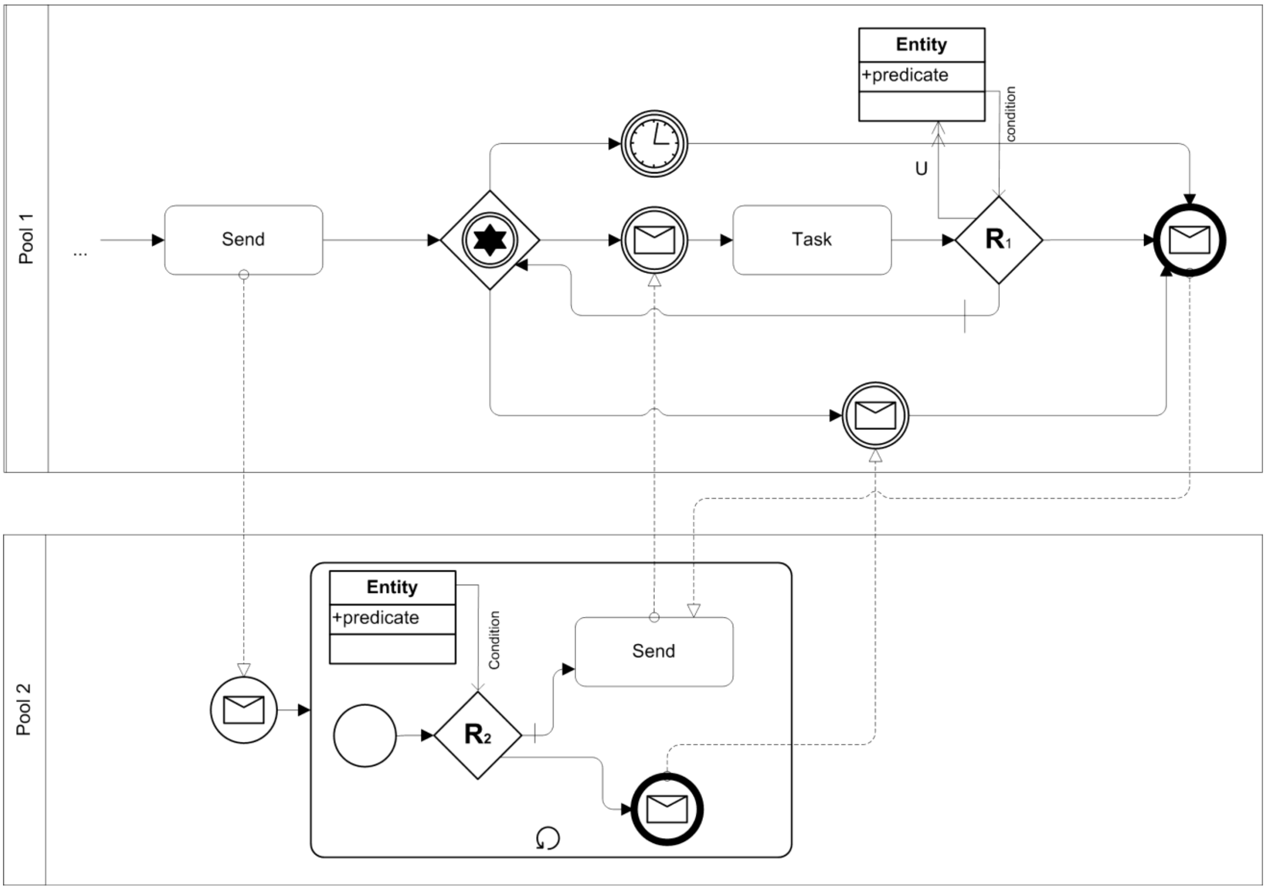


Figure 154. The “Multi-responses” pattern

The “Multi-responses” pattern represented as an interaction model in rBPMN is shown in Figure 155. In Figure 155, we show that the start message is sent from Pool 1 to Pool 2 and that after sending this message, Pool 1 expects a number of messages from Pool 2. When the message from Pool 2 has been received, the sequence flow goes from the message event to the rule gateway (it is actually rule gateway R_1 from Figure 154), which uses a reaction rule attached to it to decide whether the *stop condition* is satisfied or not. If the stop condition is satisfied, the reaction rule attached to the rule gateway updates the Entity’s predicate in order to stop any further received messages from Pool 2 and the sequence flow goes to the last message event in the process. If the reaction rule attached to the rule gateway evaluates the stop condition to false, the sequence flow goes to the event-based gateway to wait for another event. A timeout can occur also, and in that case, an intermediate timer event happens, which ends the process by connecting to the last message event. In addition, Pool 2 can send the stop message to Pool 1; this implements the stop condition on the Pool 2 side. The last message event is used to inform Pool 2 that it should not send messages any more.

In the interaction model shown in Figure 155, we do not show rule gateway R_1 from Figure 154, as we consider it as part of the internal logic of Pool 2. This is the main difference between those two models shown in Figure 154 and Figure 155. This implies that not all the process logic from the interconnection interface model can be mapped completely into the interaction model. In case of this pattern, we find that a constraint in the interaction model shown in Figure 155 is that we cannot show the logic of Pool 2 and its sending of multiple messages to Pool 1, because we would need to have repeating subprocess for this message send, but in this case the pattern would not be correct, as we need to navigate to other events from event-based gateway in this patterns too, if they happen.

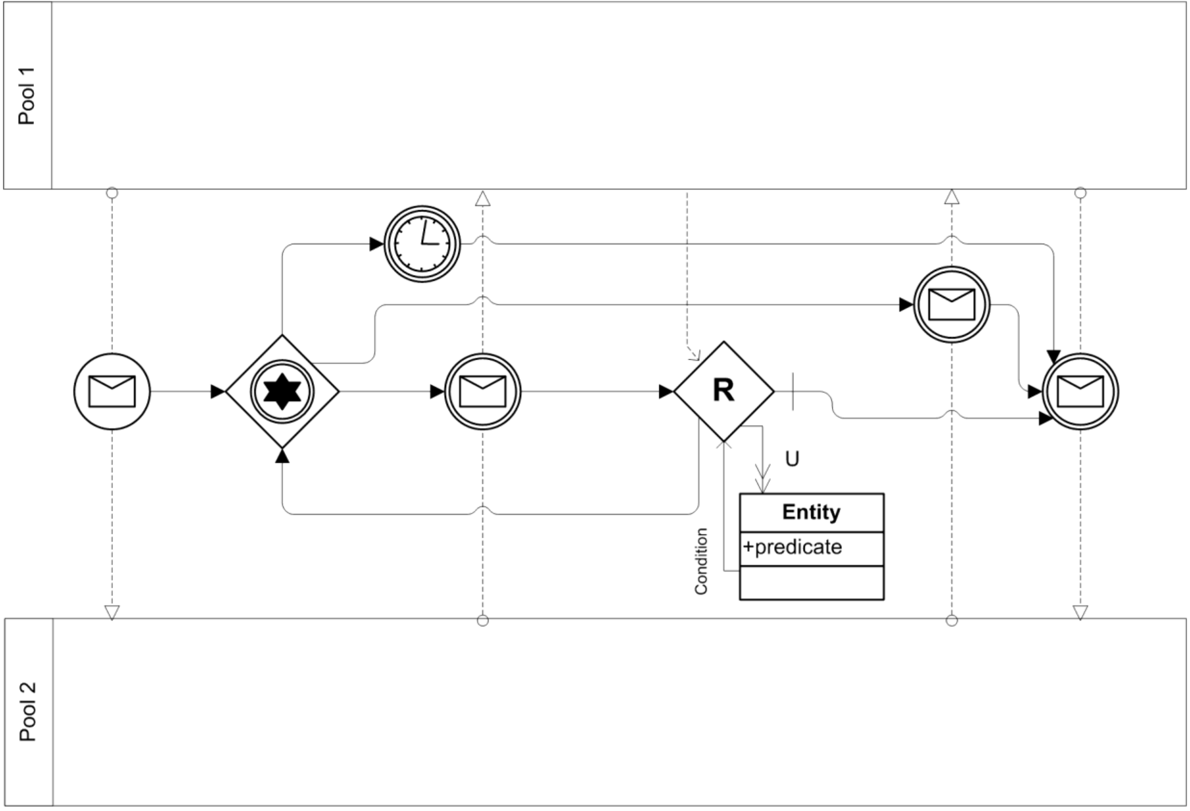


Figure 155. The “Multi-responses” pattern (interaction model)

4.2.3.2. Contingent requests

In the contingent requests pattern, a participant sends a request to another participant. If this second participant does not respond within a given period of time, the request is sent to another (third) participant. Again, if no response comes back, a fourth participant is contacted, and so on. Delayed responses, that is, responses arriving after a time-out has occurred, might or might not be discarded. In scenario shown in Figure 156, the Pool 1 is sending the message to multiple instances of Pool 2, by using the “Send”, which selects participants to which a request will be sent. The participants (instances of Pool 2) are selected from the attached participant set (<par>). The messages are received by Pool 2, which uses its own logic represented by the reaction rule attached to the rule gateway R_2 in order to decide whether to respond or not. Pool 1 waits for some amount of time for a message from Pool 2 and when such a message arrives in, Pool 1 invokes its “Task”, which is followed by a reaction rule attached rule gateway R_1 to determine if this process will end or it will return to the event-based gateway to wait for new messages. If the message from Pool 2 is not received in a given amount of time, the intermediate timer event occurs and the sequence flow is returned back to the start (the “Send” task). If a late (time-outdated) response from some earlier participant came during the processing of the contingent request (by a Pool 2 participant in Figure 156), a reaction rule attached to the rule gateway R_1 decides if such a response should be accepted or not.

In this pattern, we used a reaction rule attached to rule gateway R1 to decide whether the next participant should be contacted or not. This is an important issue for this pattern, as this selection cannot be easily represented in plain BPMN.

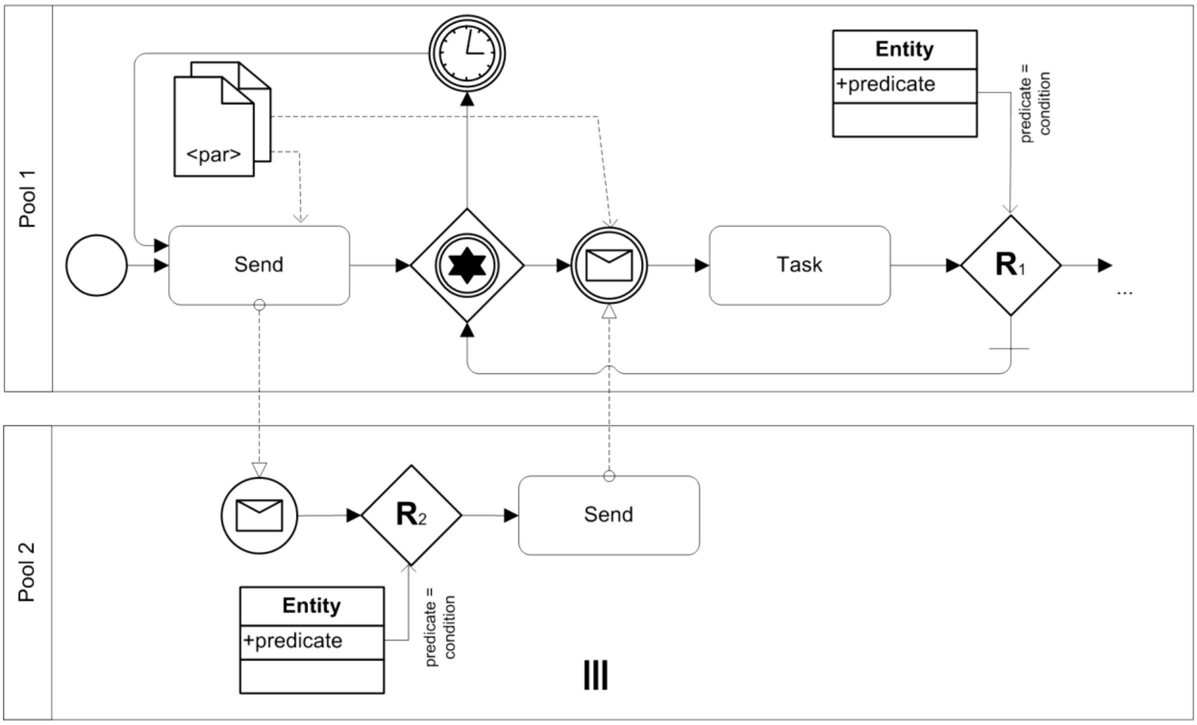


Figure 156. The “Contingent requests” pattern

The “Multi-responses” pattern represented as an interaction model in rBPMN is shown in Figure 157. In this model, we have a message that is sent on the start of the process from Pool 1 to one of the participants of the Pool 2 type, by using a reference to that participant from the participant set (<par>). Then, response messages are expected from Pool 2 in a given amount of time. When the message arrives from Pool 2, the rule gateway is used to determine whether the process will end or it will be back to wait for another message.

In the interaction model shown in Figure 157, we have a similar constraint in showing the internal logic of Pools’ 2, as in the previous pattern (Multi-responses). Therefore, because of this constraint, we can say that the interaction model cannot show a complete interaction logic as much as a behavior interconnected model can. However, this may not be a general rule, because one may not need to have any complex logic on the side of Pool 2.

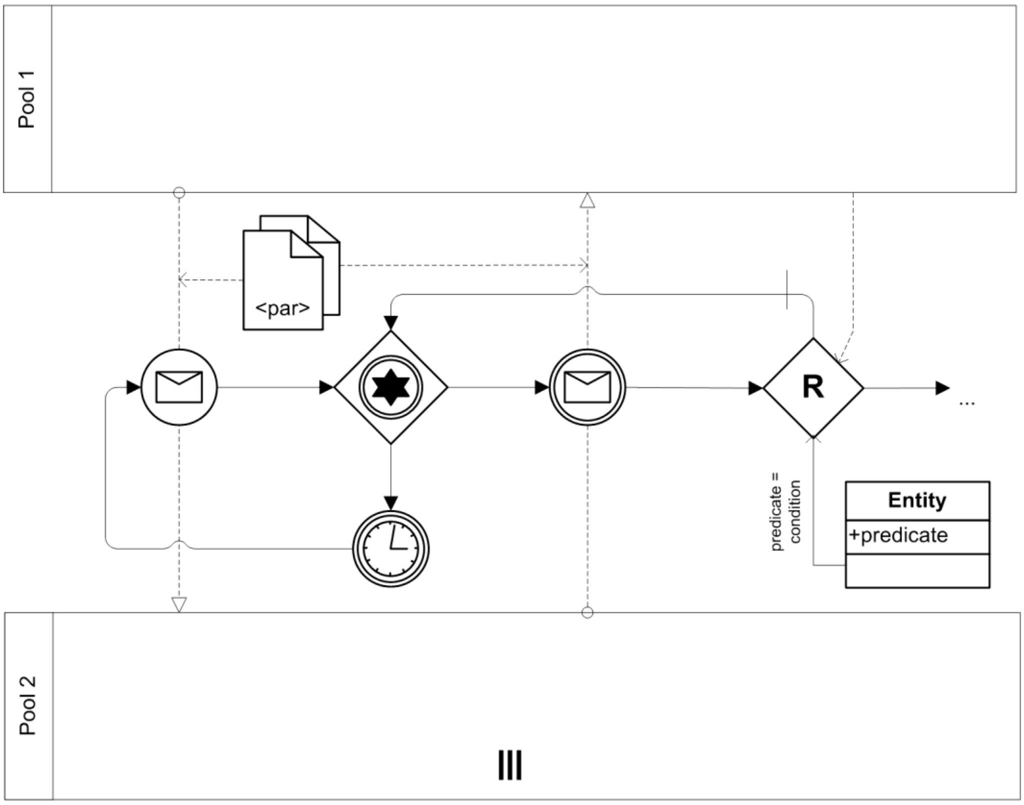


Figure 157. The „Contingent requests“ pattern (interaction model)

4.2.4. Routing patterns

In this group of patterns, there are three patterns dedicated to routed interactions, which are described in the following subsections.

4.2.4.1. Request with referral

The *request with referral* pattern is especially important in service-oriented environments where a registry is in place that allows for service binding at run time. As well, simple types of dynamic behavior can be represented by this pattern, for instance, the transmission of a new collaboration partner during an interaction.

In particular, the request with referral pattern can be used if a participant A sends a message to another participant B containing a reference to participant(s) C. Although B does not need to know all C's in advance, B can now interact with C's. This pattern describes the concept of *link passing mobility*[8]. In an rBPMN behavior interconnected model of this pattern, shown in Figure 158, the participant (pool) A uses the "Send 1" task to send a message to the pool B, which receives this message and initiates the Subprocess. In this Subprocess, based on the rule gateway (R_1) condition, if the reaction rules' condition attached to this rule gateway evaluates to true, Pool B can send messages to C pool instances, by using the "Send 2" task. In the case when the rule gateways' reaction rule condition evaluates to false, the end exception event is reached, and the Exception Handler is invoked, which uses the end message task to send the fault message to Pool A. A fault message is also generated if an error occurs during the execution of the "Send 2" task. The "Send 2" (multi-instance) task is sending messages to C pool instances, by using references to these participants from the participant set sent from Pool A. Pool C receives the message, and if the rule gateways' (R_2) attached reaction rule

[8] *Link passing mobility* is a concept that includes passing references to some participants in interaction between those participants.

condition is true, it responds to Pool A by using the “Send 3” task and a reference to Pool A from the participant set sent from Pool B.

The reaction rules attached to rule gateways enabled us not only to more precisely decide whether to send the messages to related participants, but also to support exception handling. This wouldn’t be possible in plain BPMN, and so this pattern can be only partially supported.

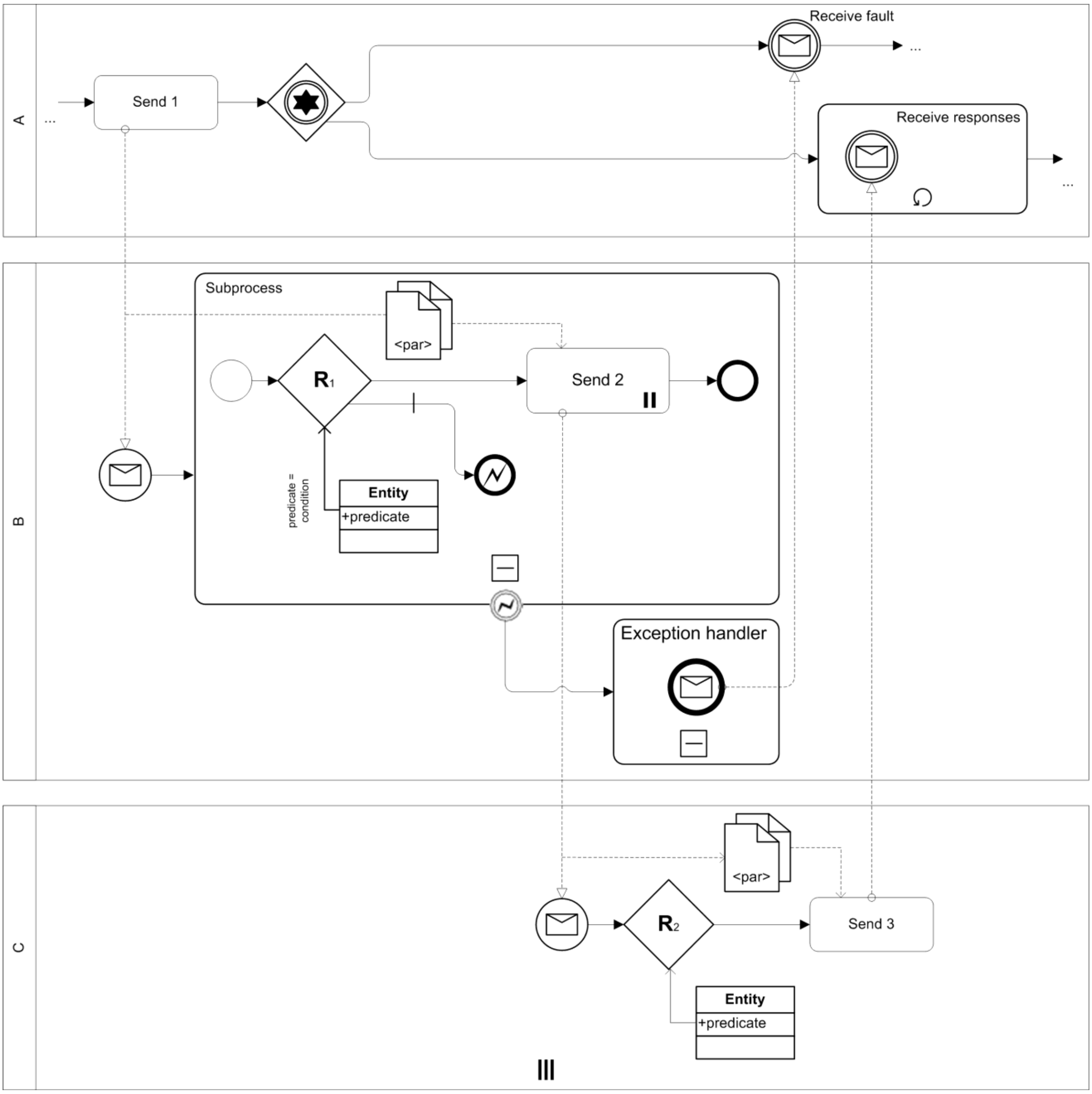


Figure 158. The “Request with referral” pattern

The “Request with referral” pattern represented as an interaction model in rBPMN is shown in Figure 159. The “Send 1” task from the interconnected behavior model, shown in Figure 158, is translated to a message event, sent from Pool A to Pool B. When Pool B receives the message, it uses the reaction rule attached to the rule gateway (R_1) to choose whether to send the message to Pool C instances, by using the “Send 2” subprocess, or to return a (fault) message to the Pool A, by using an exception event annotated message. The “Send 2” subprocess uses references to participants of Pool C, from the participant set sent from Pool A, to send messages to those participants. During the “Send 2” subprocess, an exception can occur. This exception is handled by the fault handler attached to the border of the “Send 2” subprocess, which connects to the exception event annotated message sent from Pool B to Pool A. When Pool C receives the message from Pool B, it uses the reaction rule attached to the rule gateway R_2 in a repeating subprocess, to decide whether it will respond to Pool A (by sending the message by using the “Send 3” message event), which is referenced from the participant set.

In interaction model shown in Figure 159 we completely supported interconnection model from Figure 158.

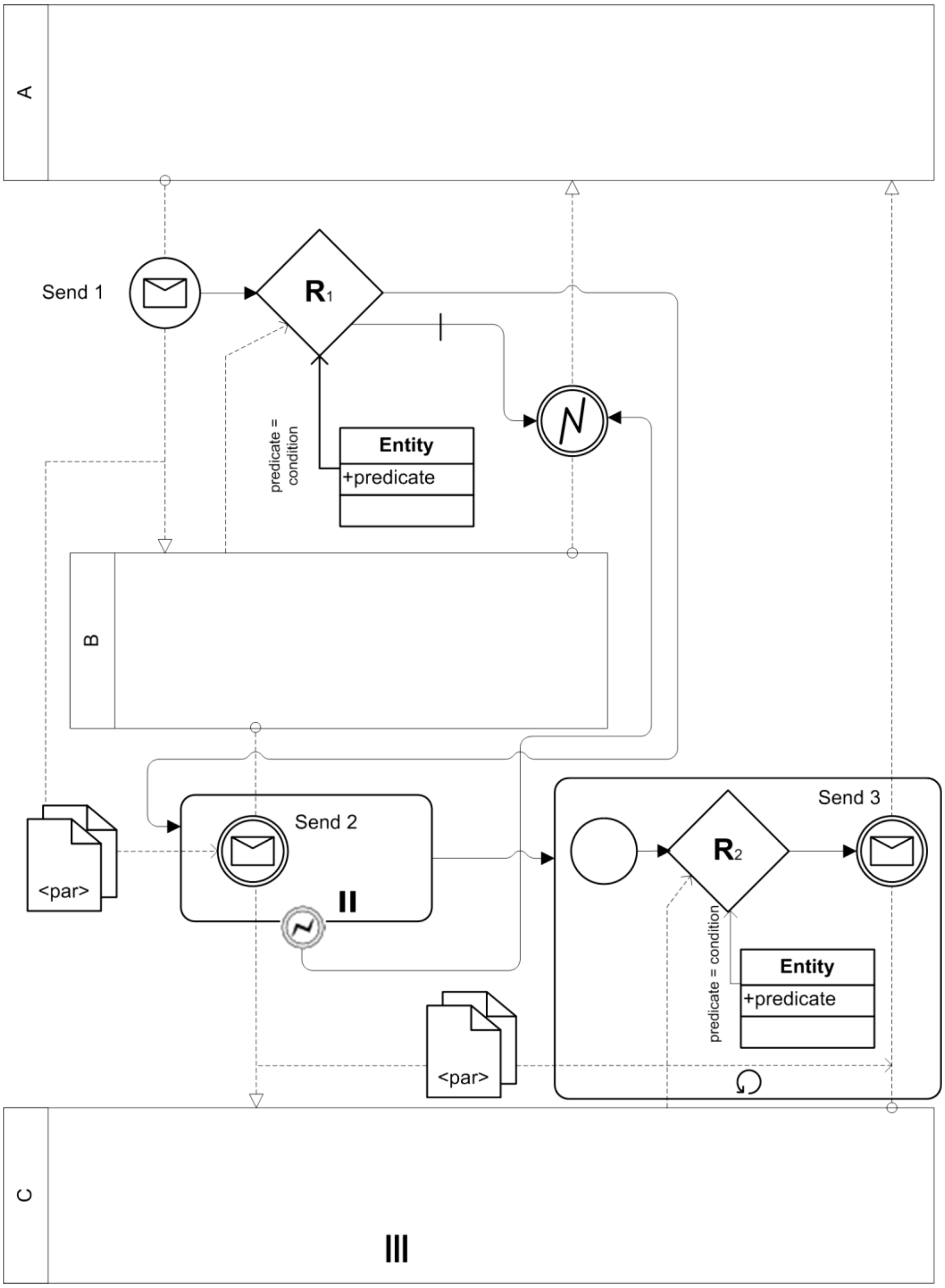


Figure 159. The “Request with referral” pattern (interaction model)

4.2.4.2. Relayed Request

The relayed request pattern is common in emailing collaboration scenarios. In this scenario, a participant A sends a request to another participant B who forwards it to a third participant C who will actually interact with A. However, B always gets copies of messages exchanged in order to be able to observe the conversation.

An rBPMN behavior interconnected model of an example of this pattern is shown in Figure 160. In the pattern example, we use reference passing between participants A and B, and B and C. When Pool A sends a message to Pool B, by using the "Send 1" task, Pool B receives this message and sends multiple messages to Pool C instances, by using references to the participants from the participant set, which are sent from Pool A. When Pool C receives the message, it uses the reaction rule attached to the rule gateway (R_1) to decide whether to send the same message to Pool A and Pool B, or to generate a fault. This fault is handled by the Exception handler, which sends the fault message to both Pools A and B. Pool A just receives the message by using the repeating Subprocess 3 with the message event and the "Receive" task, while Pool B uses a reaction rule attached to the rule gateway (R_2) when it receives the message from Pool C, to decide whether the message should be received or not. This rule-based decision is important, as Pool B needs to have a mechanism to accept only those messages that are of its particular interest.

In the standard BPMN, acceptance of particular messages would be hard to model. Additionaly, by using rules we enable that this decision can be dynamically changed in design-time or run-time.

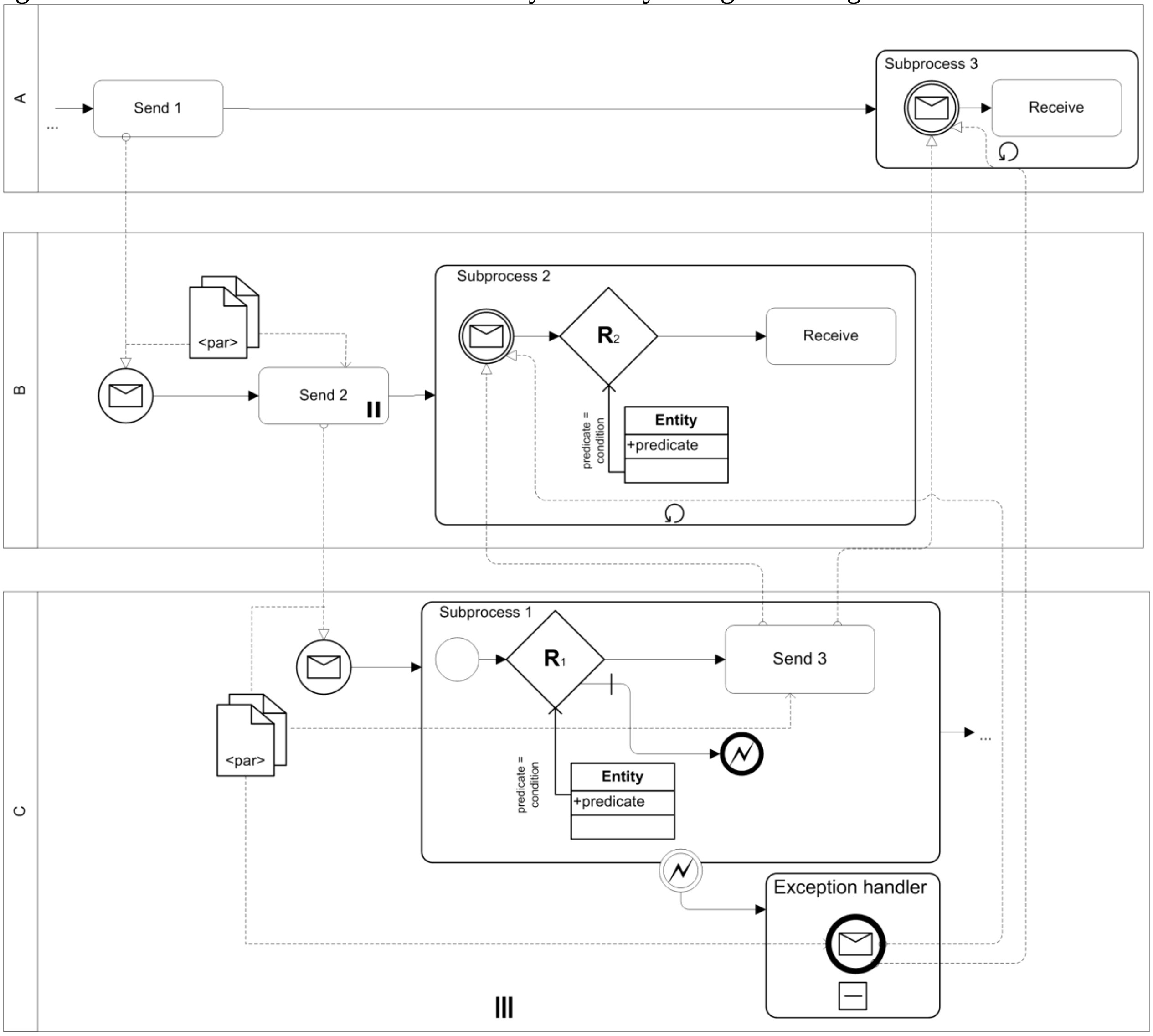


Figure 160. The "Relayed request" pattern

The "Relayed request" pattern represented as an interaction model in rBPMN is shown in Figure 161. After sending the message from Pool A to Pool B (by using the "Send 1" message event annotated flow), Pool B sends messages to the instances of Pool C by using the multi-instance subprocess. The C participants are referenced from the participant set sent from Pool A. After sending the message from Pool B to Pool C, the reaction rule attached to the rule gateway (R_1) is used to decide whether the standard or fault messages should be sent from Pool C to both Pools A and B. We have not shown the R_2 rule gateway as we consider it as a part of the Pools' B internal implementation logic, as in Multi-responses and Contingent requests patterns. This is the case, as Pool 2 should internally decide whether to receive message or not. Here we can see that interaction model cannot completely represent logic from a corresponding interconnection model.

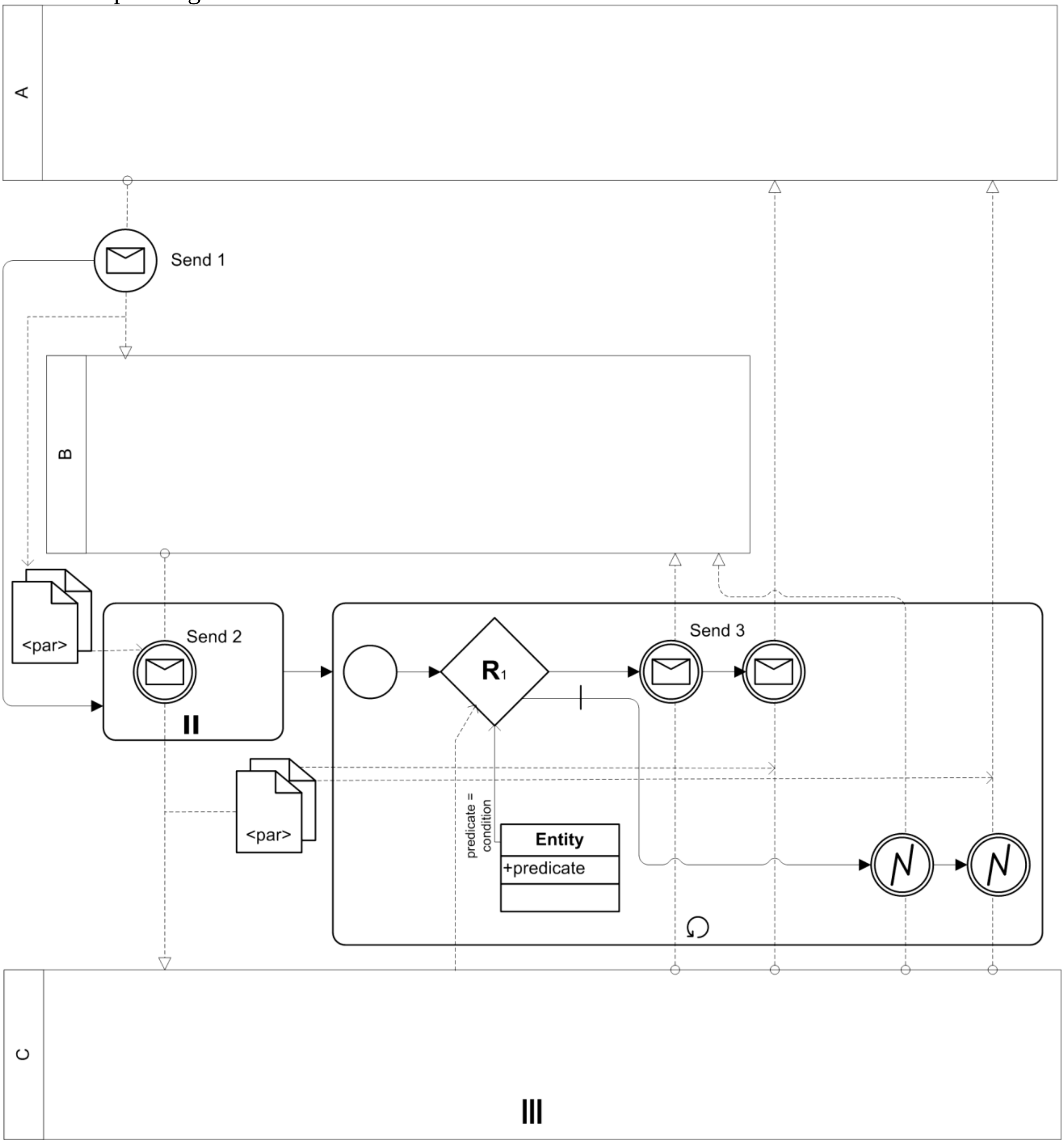


Figure 161. The "Relayed request" pattern (interaction model)

4.2.4.3. Dynamic Routing

This pattern is defined as follows [6]: “request is required to be routed to several parties based on a routing condition. The routing order is flexible and more than one party can be activated to receive a request.” This routing can be subject to dynamic conditions, which are based on data contained in the original request or obtained during the conversation. An rBPMN behavior interconnected model of this pattern is shown in Figure 162. In this pattern, Pool 1 sends a message (by using the “Send 1” task) to Pool 2, which then uses the “Send 2” task and the reference to the Pool 4 participant from the participant set to send the message to Pool 4. When Pool 4 receives the message, based on the *routing condition* defined by a reaction rule attached to the rule gateway (R), it sends the message to Pool 1 (by using the “Send 3” task) or to Pool 2 (by using the “Send 6” task). In the first case (when the reaction rules’ condition is satisfied), Pool 1 can receive the message, process it by using the “Send 4” task and return it to Pool 2. Then, Pool 2 uses the reference to Pool 4 and the “Send 5” task to return the message to Pool 4. Pool 4 uses the “Receive” task to receive the message. In the second case (when the reaction rules’ condition is not satisfied), Pool 4 uses the “Send 6” task and the reference to the Pool 3 participant to send the message to this participant. When Pool 3 receives the message, it forwards it to Pool 1 by using the “Send 7” task and the reference to Pool 1 from the participant set.

We have partially supported this pattern, as it define that a participant can insert new or delete existing interactions in choreography at runtime. This is not possible, as BPMN semantics would be broken in that case, because message flows cannot be dynamically added or removed from a choreography. In this pattern, without usage of rules, it would be hard to support routing condition.

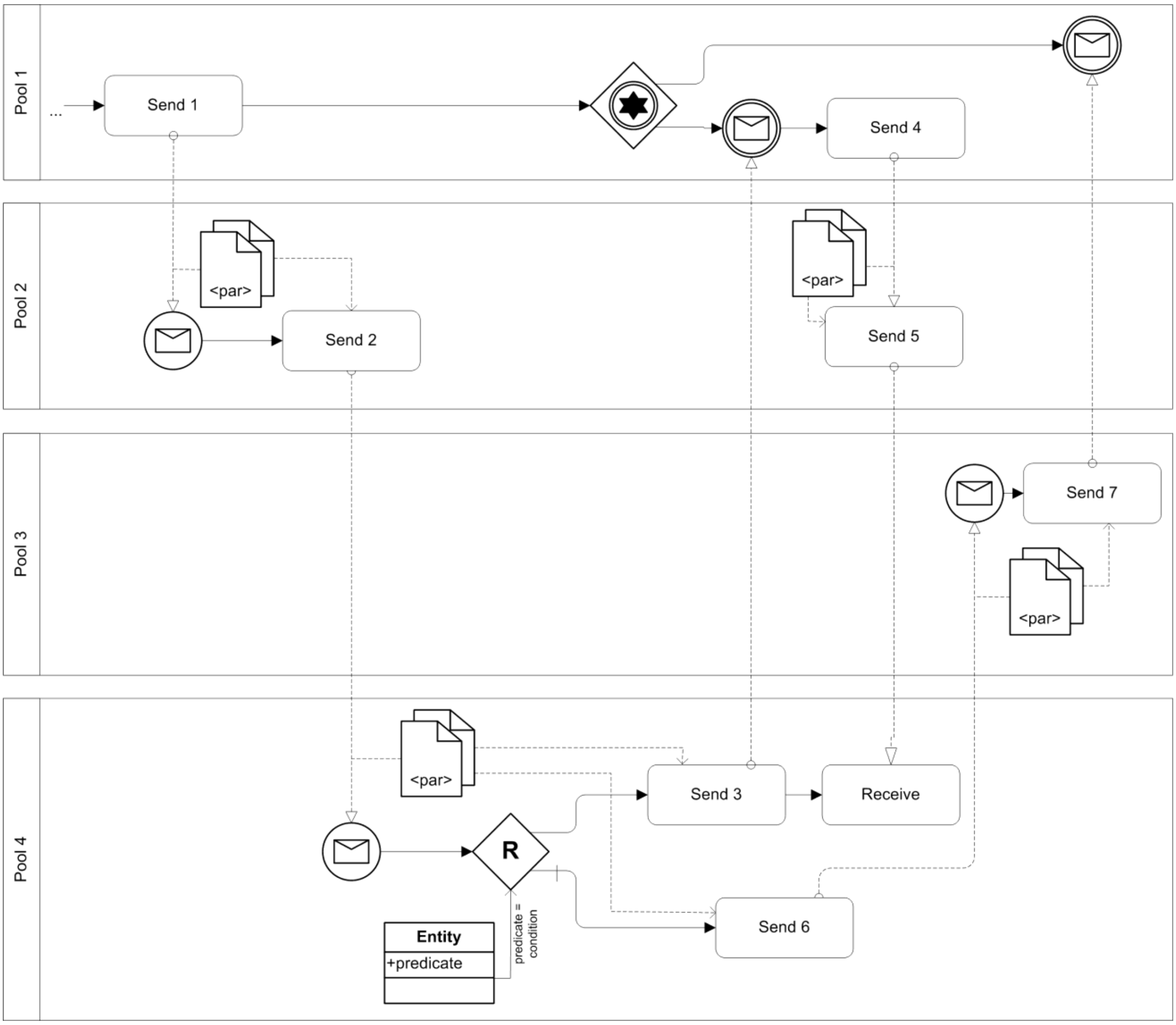


Figure 162. The "Dynamic routing" pattern

The "Dynamic routing" pattern represented as an interaction model in rBPMN is shown in Figure 163. Pool 1 is using the "Send 1" message event to send the message to Pool 2, and then Pool 2 sends the message to Pool 4 by using the "Send 2" message event and the reference to Pool 4 from the participant set. When Pool 4 receives the message, it uses the rule gateway to evaluate the routing condition. The routing condition is used to decide whether the reaction rule attached to the rule gateway will use the "Send 3" message event to send the message to Pool 1, or the "Send 6" message event to send the message to Pool 3. When the message is sent to Pool 1, Pool 1 uses the "Send 4" task to return the message to Pool 2. After sending this message, Pool 2 uses the "Send 5" message event and the reference to Pool 4, to send the message to Pool 4. When the reaction rule attached to the rule gateway evaluate its condition to false, the "Send 6" message event and a reference to Pool 3 from the participant set is used to send the message from Pools 4 and 3. Then, Pool 3 sends the message to Pool 1 by using the "Send 7" message event and a reference to the Pool 1 from the participant set.

In interaction model shown in Figure 163 we mapped completely corresponding interconnection model from Figure 162.

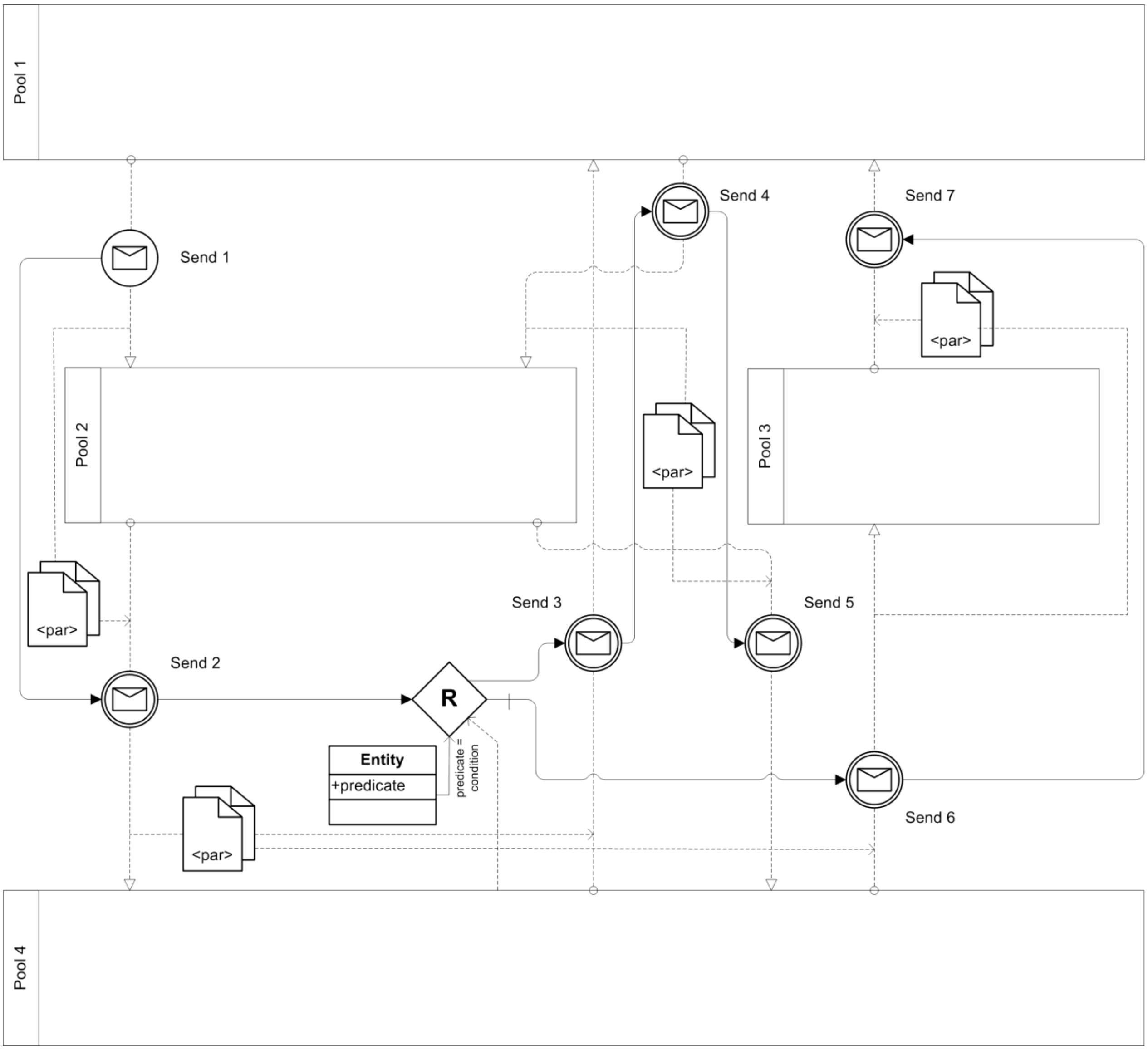


Figure 163. The "Dynamic routing" pattern (interaction model)

4.2.5. Mappings between rBPMN interconnection and interaction models

In this subsection, we present how rBPMN interconnected behavioral interface models (orchestration) can be generated from rBPMN interaction models (choreographies). This is an imporant issue, as we want to enable generation of multiple interconnection models from one interaction model. With this solution, we can assign adequate process logics to each involved participant in a process, from one global interaction model.

Our approach is based on work on deriving interface behavior models from iBPMN choreographies [22] by using Petri nets as intermediary and applying to it model reduction algorithm [150]. This algorithm removes these interactions from the model, in which the corresponding participant is not involved. In this section, we propose a mapping of rBPMN interconection models from rBPMN interaction models, based on work presented in [22] and our research presented in previous section (service interaction patterns).

The first four mappings are based on solution proposed in [22]. In Figure 164, we show the start event message mapping. In Figure 164a, we show an interaction model where pool A sends a message (annotated with a start message event) to pool B. In Figure 164b, we have the same model as an interconnection model, where the start message event is sent from pool A is mapped into a sending task in pool A. This mapping is realized in this way, because sending a message from one pool to another can be realized by using a task inside a pool. After the start message event in Figure 164a, the sequence flow is continued and this is denoted with *m*, which represents any other element in a sequence flow. This interaction is shown in multiple Service Interaction Patterns such as Send, Receive, and Multi Responses.

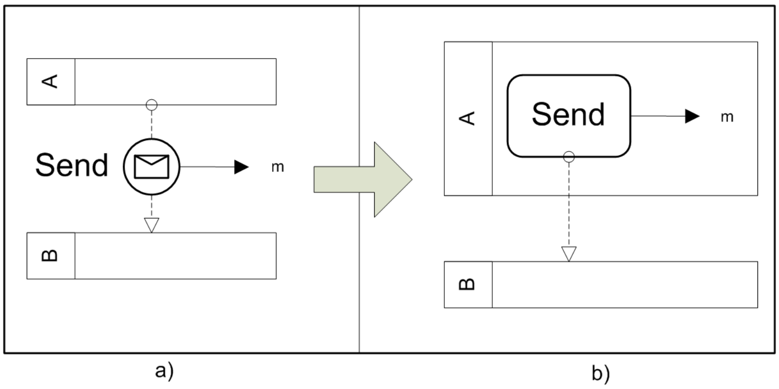


Figure 164. The "Start event message" mapping

The Intermediate event message mapping shown in Figure 165 is very similar to the Start event message mapping, but in this case, the message event have an incoming sequence flow (denoted with m) as well as an outgoing sequence flow (denoted with n) [22]. Based on the previous mapping, we mapped these two sequence flows as incoming and outogoing sequence flows of the sending task in pool A of Figure 165b. This interaction is shown in different Service Interaction Patterns such as Send/Recieve, and Racing Incoming Messages.

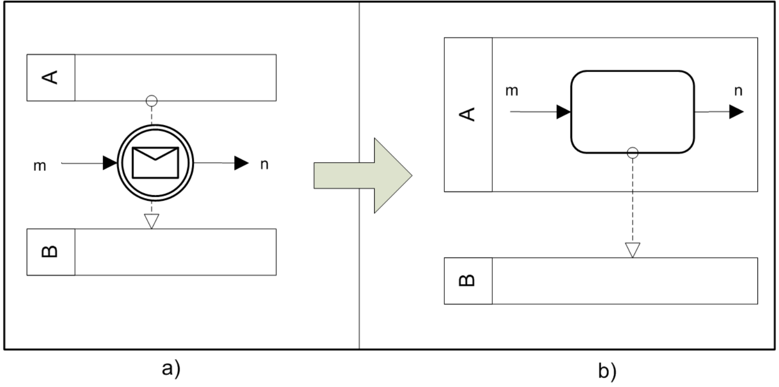


Figure 165. The "Intermediate event message" mapping

The next mapping is shown in Figure 166. Here, we show the "Parallel split" mapping, where in the interaction model (Figure 166a), a gateway is connected to pool A. This means that pool A carries out the choice. This mapping is shown in Figure 166b where the gateway is located in pool A, because if A carries out the choice, then it is the corresponding participant. The "Parallel split" mapping is present in all the Service Interaction Patterns, where the rule gateway is used as shown in Section 4.2.

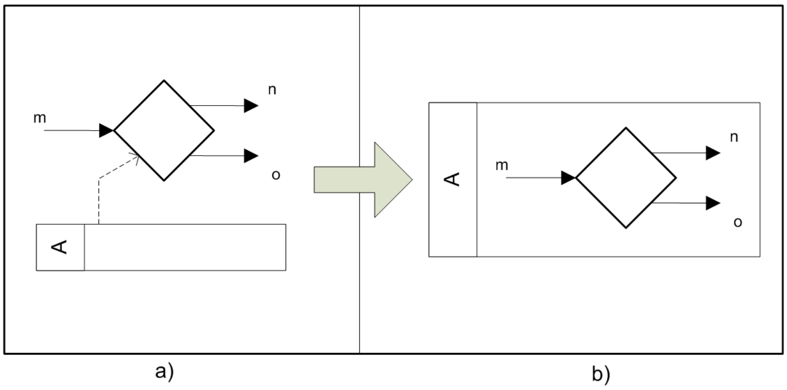


Figure 166. The “Parallel split” mapping

Similar to the previous mapping, by following the mappings presented in [22], in Figure 167, we define the “Synchronization” mapping. In the interaction model of this pattern (Figure 167a), an intermediate message event is attached to the message which is sent from pool A to pool B. The sequence flow which continues to the gateway, is denoted with n or o (Figure 167b). The gateway will be located in pool A, because it is connected with pool A (Figure 167a). The interconnection model shown in Figure 167a is mapped into a Send task, which sends the message and which is connected with the gateway (Figure 167b). This mapping is based on the “Intermediate event message” mapping. This interaction is used in different Service Interaction Patterns such as Racing Incoming Messages.

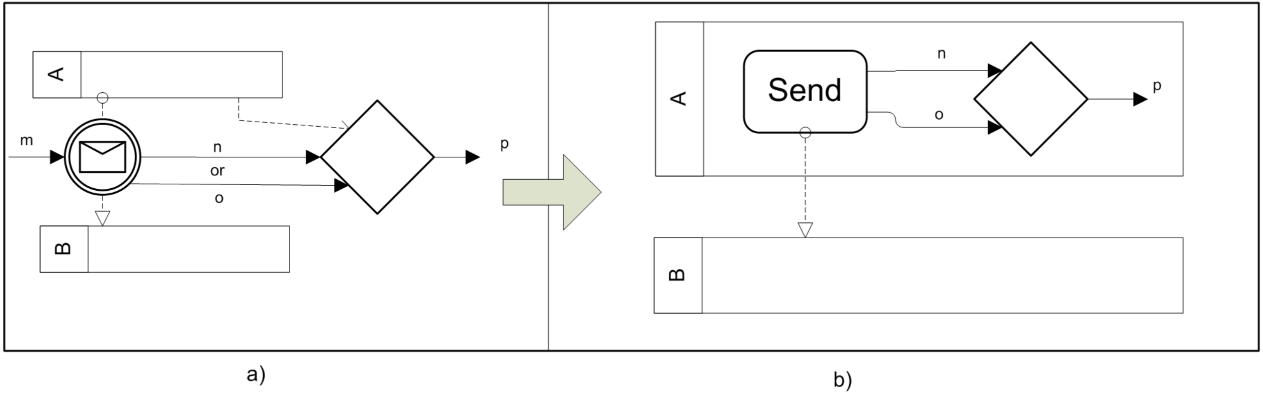


Figure 167. The “Synchronization” maping

In Figure 168, we show the “Multiple Instances Task” mapping. Figure 168a shows an interaction model in which pool A sends a message to multiple instances of pool B. The Message event, which sends the message from pool A to pool B, is located in the multiple instance send subactivity (Figure 168a). This pattern is mapped into the interconnection model, by using the multiple instances Send task, which sends the message to the multiple instances of pool B (Figure 168b). Multi-instance pool B is denoted by a small parallel indicator (|||) displayed at the bottom-center of the pool. This mapping is present in different Service Interaction Patterns, such as in the One to Many Send pattern.

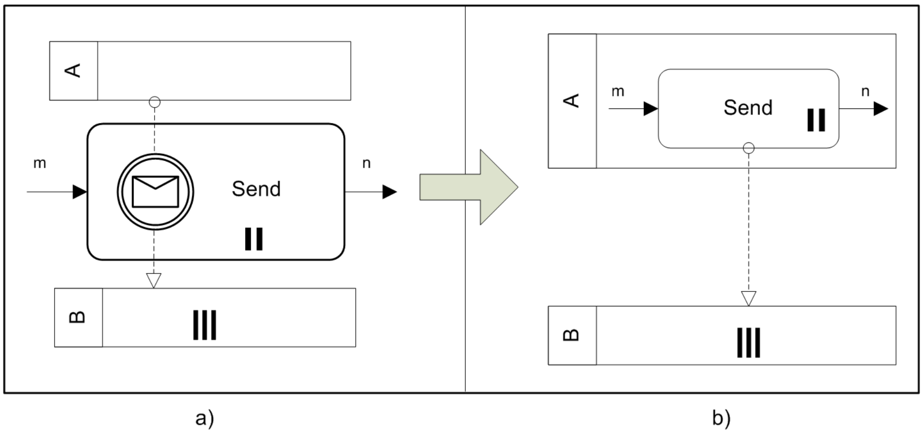


Figure 168. The “Multiple Instances Task” mapping

The following two patterns are related to the participant sets. In Figure 169a, we show the modified “Start event message” mapping, where the participant (multi-instance pool B) to whom the message should be sent is taken from a corresponding Participant set, which the message flow points to. This interaction is mapped to the Send task in Figure 169b, following the Start event message mapping, where the Participant set points to this sending task. This mapping exists in multiple Service Interaction Patterns, such as the Send/Receive and One to Many Send patterns.

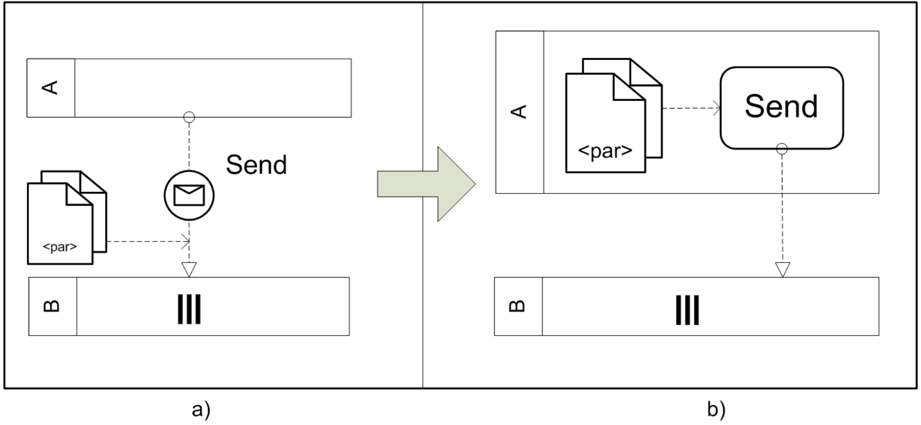


Figure 169. The “Participant set sends message” mapping

In Figure 170, we show the “Participant set receives message” mapping where in the interaction model (Figure 170a), the participant information is passed through a message sent from pool A to pool B. This is mapped to the corresponding interconnection model in Figure 170b, where the participant information is received from a message received in pool B, and this information is passed to the Receive task, which uses this information for further actions. This mapping exists in different Service Interaction Patterns, such as Relayed Request and Dynamic Routing.

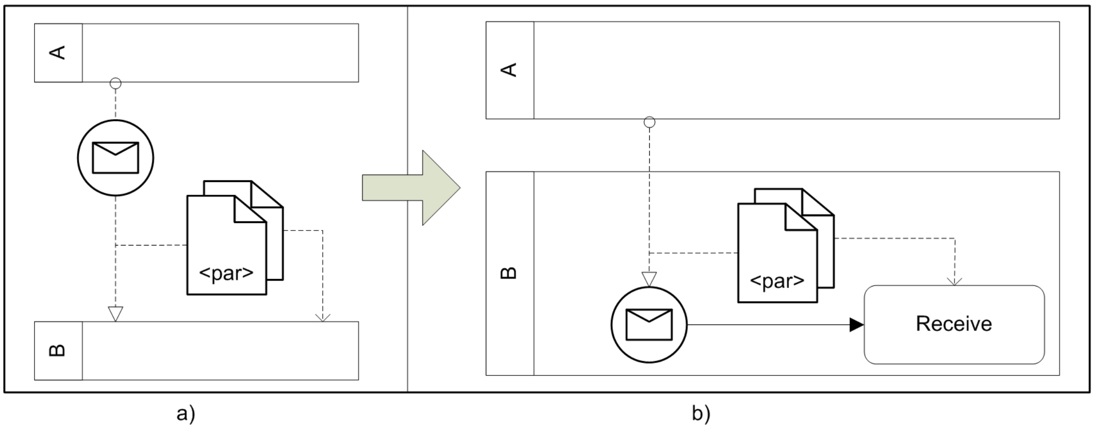


Figure 170. The “Participant set receives message” mapping

4.3. Modeling control flow in rBPMN

The control flow patterns represent a set of 21 workflow patterns created to show expressivity of workflow management systems [116]. These patterns can be used in business process modeling, and to compare the expressiveness of business process languages. The basic control flow patterns include sequence, and split, and join, as well as exclusive or split and exclusive or join. Control flow patterns are defined at the process model level.

As these patterns can be presented in the standard BPMN without using rules, by adding rules we enrich the BPMN models in a way that such business processes can be more precisely described in a declarative way and changed in a design-time or in a real-time by changing only rules without a need to redesign the whole process. By employing rules in a process allows us to dynamically change the control flow of a process. We also use these patterns to evaluate where in a workflow process our rules can be used. As BPMN [88] has a weak support for rule-based gateways, where conditions are usually written in a natural language [108], by adding formal rules, we enable execution of such processes possible on some execution platform, such as BPEL [49].

We should note that these patterns apply to business models that are used to model process orchestrations, because activities used in these patterns are performed within a single organization (i.e., BPMN Pool). Regarding mappings from rBPMN to execution languages, such as BPEL, these business models can be mapped into BPEL and WSDL constructs by using already defined mappings [88] [147], where rules are can mapped by using standard BPEL constructs (such as invoke) or into the executable rules that extend BPEL.

The control flow patterns represent a set of 21 workflow patterns created to show expressivity of workflow management systems [116]. The patterns in this section came from the research done in a Workflow Patterns Initiative[9]. Authors [116] recognized patterns organized in the following groups: basic control flow patterns, advanced branching and synchronization patterns, structural patterns, multiple instance patterns, state-based patterns and cancellation patterns. We follow this organization and order of the patterns in the rest of the section.

4.3.1. Basic Control Flow Patterns

The patterns in this group include elementary aspects of process control.

[9] See http://www.workflowpatterns.com for more details.

4.3.1.1. The "Sequence" Pattern

The pattern "Sequence" is represented in Figure 171. An activity of the type *activity2* is started after the completion of an activity of the type *activity1*. This pattern represents an ability to show a sequence of activities. We used the reaction rule attached to the rule gateway in this pattern to define a condition under which the sequence flow will continue from *activity1* to *activity2*, or the sequence flow will stop. In the standard BPMN, this pattern is modeled without the rule gateway. This rule gateway, however, enables us to define more precisely the condition under which the sequence of activities continues. In addition, the reaction rule condition can be updated both at run- and design-time of the workflow.

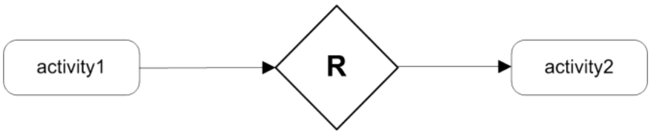


Figure 171. The "Sequence" pattern

To ensure that a rule connected to the rule gateway in this pattern is a reaction rule, and that it have *activity1* for its *triggeredEventExpr* and *activity2* for *triggeringEventExpr*, we define the following OCL constraints to enforce this relation between the elements of the metamodel. In fact, this constraint has been defined on the corresponding rBPMN metamodel elements (as described in Section 3.2):

```
[1] The rule gateway is connected to the reaction rule

context RuleGateway
inv: rule->exists(e | e.oclIsKindOf(ReactionRule))

[2] The reaction rule attached to the rule gateway needs to have the same task for
its triggeredEventExpr, as the rule gateway’s outgoing task, and the same task for
its triggeringEventExpr, as the rule gateway’s incoming task.

context RuleGateway
inv: let InSequenceFlow : SequenceFlow.allInstances()->select(c | c.targetRef
                              = this)->asSequence()->first() in
          let OutSequenceFlow : SequenceFlow.allInstances()->select(c |
                                 c.sourceRef = this)->asSequence()->first() in
             rule->first().triggeringEventExpr.task = InSequenceFlow.sourceRef
                     and rule->first().triggeredEventExpr.task =
                              OutSequenceFlow.targetRef
```

4.3.1.2. The "Parallel Split" Pattern

The pattern "Parallel Split" splits an activity into two or more activities which can be performed in parallel, thus allowing activities to be performed simultaneously or in any order. This pattern is shown in Figure 172. In Figure 172, after the end of an *activity1*, activities of the type *activity2* … *activityn* are started being performed in parallel. In this case the *triggeredEventExpr* of the reaction rule attached to the rule gateway is R2ML *ParallelEventExpression* that contains two or more activities (i.e., *activity2* ... *activityn*). In the standard BPMN, this pattern could be modeled with an AND-gateway, implicitly or through sub-Activities [148], but the advantage of using rules here is that we can define a condition on which parallel split could happen.

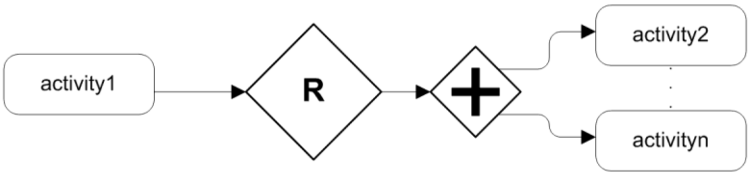


Figure 172. The “Parallel Split” pattern

For this pattern, we have defined following OCL constraints:

```
[1] The reaction rule attached to the rule gateway for its triggeringEventExpr need
to have a task, while for its triggeredEventExpr need to have a
ParralelEventExpression of two or more activities.
context RuleGateway
inv: rule->first().triggeredEventExpr.oclIsKindOf(ParallelEventExpression)
           and rule->first().triggerringEventExpr.OclIsKindOf(Task)

[2] The reaction rule attached to the rule gateway needs to have the same parallel
tasks for its triggeredEventExpr, as the rule gateways’ outgoing tasks, and the
same task for its triggeringEventExpr, as the rule gateways’ incoming task.

context RuleGateway
inv: let InSequenceFlow : SequenceFlow.allInstances()->select(c | c.targetRef
                              = this)->asSequence()->first() in
     let OutSequenceFlow : SequenceFlow.allInstances()->select(c |
                                c.sourceRef = this)->asSequence()->first() in
        let OutActivites : SequenceFlow.allInstances->select(c | c.sourceRef =
                                   OutSequenceFlow.targetRef)->asSequence()
                                             ->collect(e | e.targetRef) in
       rule->first().triggeringEventExpr.task = SequenceFlow.sourceRef
           and
       rule->first().triggeredEventExpr->forAll(p | OutActivities->exists(p))
```

4.3.1.3. The “Synchronization” Pattern

The pattern “Synchronization” merges two or more parallel activities into one activity. An example is presented in Figure 173. In the figure, after all parallel activities of the types *activity1 … activityn-1* have ended, an activity of the type *activityn* is started. This pattern in the standard BPMN could be represented with an AND-gateway or partially through sub-Activities [148]. By using the reaction rule attached to the rule gateway, we can decide under which condition parallel activities will merge into one. This condition could be changed at run- or design-time.

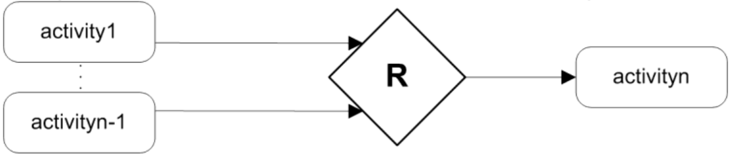


Figure 173. The “Synchronization” pattern

For this pattern, we have defined following OCL constraints:

```
[1] The reaction rule attached to the rule gateway for its triggeringEventExpr
needs to have a ParallelEventExpression of two or more activities, while for its
triggeredEventExpr needs to have a Task.
context RuleGateway
inv: rule->first().triggeredEventExpr.oclIsKindOf(Task)
   and rule->first().triggerringEventExpr.oclIsKindOf(ParallelEventExpression)

[2] The reaction rule attached to the rule gateway need to have the same parallel
tasks for its triggeringEventExpr, as the rule gateways’ outgoing tasks, and the
same task for its triggeredEventExpr, as the rule gateways’ incoming task.
```

```
context RuleGateway
inv: let InSequenceFlows : SequenceFlow.allInstances()->select(c | c.targetRef
                              = this)->asSequence()in
       let OutSequenceFlow : SequenceFlow.allInstances()->select(c |
                                c.sourceRef = this)->asSequence()->first() in
         rule->first().triggeredEventExpr.task = OutSequenceFlow.targetRef
           and
             rule->first().triggeringEventExpr.eventExpression->forAll(c |
                           InSequenceFlows->exists(e | e.sourceRef = c) )
```

4.3.1.4. The "Exclusive Choice" Pattern

The pattern "Exclusive Choice" chooses one of several activities for performing based on a control data. In an example of Figure 174, after the end of an *activity1*, if the condition specified on a reaction rule attached to the rule gateway R by the *predicate* and *condition* is true, an *activity2* is started to execute. Otherwise, *an activity3* is started. The negative choice is denoted with a crossed line between R and *activity3* in Figure 174. It should be noted that we used the crossed BPMN outgoing sequence flow from a gateway, which is chosen when the condition of the attached rule(s) evaluates to false. This means that we must have exactly two reaction rules attached to the rule gateway, where the condition is the same, only negated on the second rule. When the first rule condition evaluates to true, the non-crossed outgoing sequence flow is chosen, and when it evaluates to false, the crossed sequence flow is chosen (*activity3* in this case). This pattern shows the main advantage of using rules to choose a flow in a process, as the rule's condition can be changed in run-time, but also in design-time. If this pattern were realized in the standard BPMN, with an AND-gateway, through sub-Activities or in a context [148], this advantage would not exist.

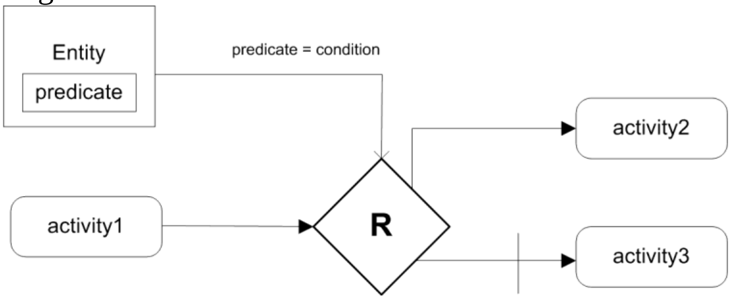


Figure 174. The "Exclusive Choice" pattern

For this pattern, we have defined the following OCL constraints:

```
[1] The rule gateway must have exactly two reaction rules attached to it.

context RuleGateway
inv: rule->size() = 2

[2] One of the outgoing sequence flows must have "Default" value for its
ConditionType. We use "Default" mark on a crossed sequence flow from a gateway.

context RuleGateway
inv: let OutSequenceFlow : SequenceFlow.allInstances()->select(c |
                 c.sourceRef = this)->asSequence()->first() in
                      OutSequenceFlow->exists(e | e. ConditionType = "Default")

[3] The triggeringEventExpr and triggeredEventExpr of both reaction rules attached
to the rule gateway must be of the Task type.

context RuleGateway
```

```
inv: rule->first().triggeredEventExpr.oclIsKindOf(Task) and
      rule->first().triggeringEventExpr.oclIsKindOf(Task) and
         rule->last().triggeredEventExpr.oclIsKindOf(Task) and
            rule->last().triggeringEventExpr.oclIsKindOf(Task)
```

[4] The reaction rule whose condition is not negated must have the same *Task* for its *triggeringEventExpr* as the rule gateway’s incoming Task, and for its *triggeredEventExpr*, the reaction rule must have the same *Task* as the rule gateway’s outgoing (default) *Task*. In addition, the reaction rule whose condition is negated, must have the same *Task* for its *triggeringEventExpr* as the rule gateway’s incoming Task, and for its *triggeredEventExpr* must have the same *Task* as the rule gateway’s outgoing (non-default) *Task.*

```
context RuleGateway
inv: let InSequenceFlow : SequenceFlow.allInstances()->select(c | c.targetRef
                              = this)->asSequence()->first() in
       let OutSequenceFlowNonDefault : SequenceFlow.allInstances()->select(c |
                                c.sourceRef = this)->select(e | e.
                                    ConditionType <> “Default”)->first() in
         let OutSequenceFlowDefault : SequenceFlow.allInstances()->select(c |
                                c.sourceRef = this)->select(e | e.
                                    ConditionType = “Default”)->first() in
            let NegatedRule : rule->select (c | c.conditions->
                                       first().isNegated = “true”)->first() in
               let NonNegatedRule : rule->select (c | c.conditions->
                                      first().isNegated = “false”)->first() in



  NegatedRule.triggeringEventExpr.task = InSequenceFlow.sourceRef
      and
  NonNegatedRule.triggeringEventExpr.task = InSequenceFlow.sourceRef
      and
  NegatedRule.triggeredEventExpr.task = OutSequenceFlowDefault.targetRef
      and
  NonNegatedRule.triggeredEventExpr.task = OutSequenceFlowNonDefault.targetRef
```

4.3.1.5. The “Simple merge” Pattern

The “Simple Merge” pattern starts executing an activity as soon as any of the preceding alternative activities ends. It is an assumption of this pattern that none of the alternative branches is ever executed in parallel. If this is not the case, the pattern “Multiple merge” or “Discriminator” is applied instead. In Figure 175, an activity of the type *activityn* is started when one preceding activity out of alternative activities of the types *activity1* … *activityn* ends. An activity of the type *activityn+1*, thus is performed only once. In order to represent this pattern, we introduce an XOR gateway, followed by the reaction rule attached to the rule gateway, used to determine which activity would be chosen to trigger *activityn+1*. In this way, we enabled that the condition of the gateway, which selects the triggering activity, could be changed dynamically, instead of the static merge in the standard BPMN done by using only the XOR gateway [148].

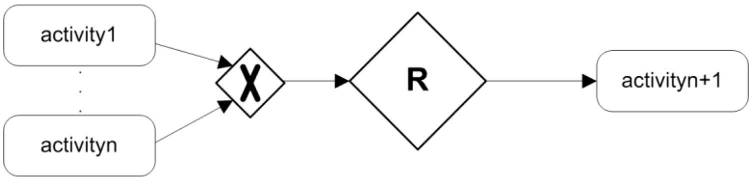


Figure 175. The “Simple merge” pattern

For this pattern, we have defined following OCL constraints:

[1] The reaction rule attached to a rule gateway for its *triggeringEventExpr* must have *ChoiceEventExpression*, while for its *triggeredEventExpr* must have a *Task*.

```
context RuleGateway
inv: rule->first().triggeredEventExpr.oclIsKindOf(Task) and
      rule->first().triggeringEventExpr.oclIsKindOf(ChoiceEventExpression)
```

[2] The reaction rule attached to a rule gateway, for its *triggeredEventExpr*, which is the same as the outgoing *Task* of the rule gateway. This reaction rule also must have the same *Tasks* in its *ChoiceEventExpression*, as the incoming *Tasks* of the rule gateway.

```
context RuleGateway
inv: let InSequenceFlow : SequenceFlow.allInstances()->select(c | c.targetRef
                              = this)->asSequence()->first() in
       let OutSequenceFlow : SequenceFlow.allInstances()->select(c |
                                c.sourceRef = this)->asSequence()->first() in
         let InActivites : SequenceFlow.allInstances->select(c | c.targetRef =
                                   InSequenceFlow.sourceRef)->asSequence()
                                             ->collect(e | e.sourceRef) in
        rule->first().triggeredEventExpr = OutSequenceFlow.targetRef and
        rule->first().triggeringEventExpr->forAll(p | InActivities->exists(p))
```

4.3.2. Advanced Branching and Synchronization Patterns

In this section, we present patterns that characterize more complex branching and merging concepts, which arise in a business process.

4.3.2.1. The "Multiple Choice" Pattern

The pattern "Multiple Choice" is the generalization of the pattern "Exclusive Choice". Based on a control data, this pattern chooses one or more activities to perform. In Figure 176, according to the reaction rule attached to the rule gateway (R1), after the end of an activity of the type *activity1*, if the condition specified by *predicate1* and *expression1* evaluates to true, an activity of the type *activity2* is started. According to the reaction rule attached to the rule gateway Rn, if the condition specified by *predicaten* and *expressionn* is true, an activity of the type *activityn* is started. Either an activity of one of the types *activity2* ... *activityn* or any combination of them in parallel may thus be performed. An activity may get performed one or more times, depending on the condition specified by the given expression and predicate, after which the branch continues without synchronizing with the other branches. As with the *Exclusive Choice*, we need to ensure that at least one outgoing branch is selected, so we also employ a default outgoing sequence flow from the inclusive gateway. The usage of rules in this pattern is important, as the main context criterion for this pattern that the information needed to calculate the logical conditions on the each of the outgoing branches is available at runtime, so by using rules and data from the vocabulary, this data could be changed dynamically (i.e., at run-time) or at design-time. This could not be realized in the standard BPMN only with the OR or Complex gateways, or without a gateway, but by using only condition sequence flows [148]. This reason for this in the standard BPMN is in the fact that such defined conditions are static in a process model.

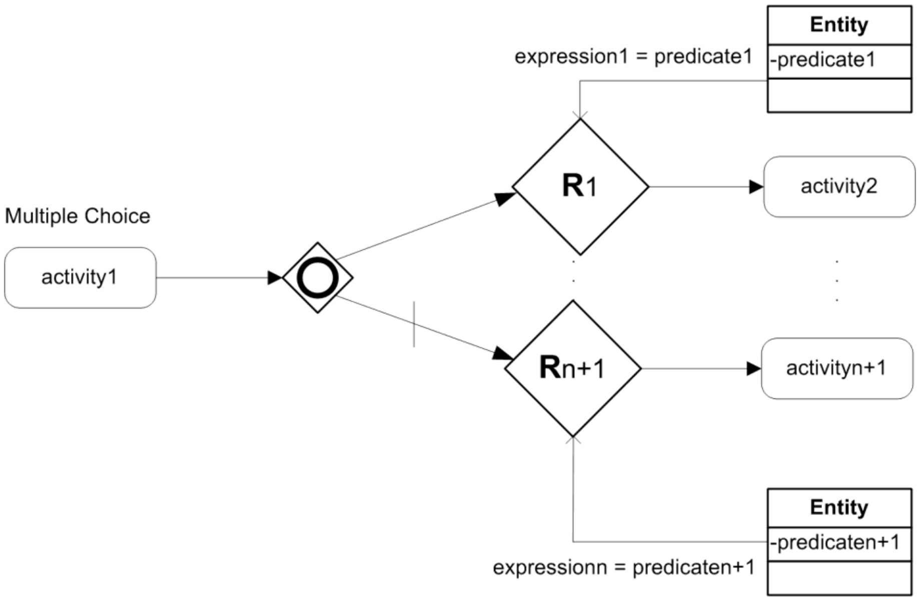


Figure 176. The “Multiple Choice” pattern

For this pattern, we have defined following OCL constraints:

```
[1] Each outgoing sequence flow from the Inclusive gateway must be followed by the
rule gateway, which have a reaction rule attached to it.

context InclusiveGateway
let OutSequenceFlow : SequenceFlow.allInstances()->select(c |
                 c.sourceRef = this)->asSequence()->first() in
      OutSequenceFlow->forAll(e | e.targetRef.oclIsKindOf(RuleGateway))

[2] The one of the outgoing sequence flows must have a “Default” value for its
ConditionType.

context InclusiveGateway
inv: let OutSequenceFlow : SequenceFlow.allInstances()->select(c |
                 c.sourceRef = this)->asSequence()->first() in
                      OutSequenceFlow->exists(e | e. ConditionType = “Default”)

[3] The triggeringEventExpr of each reaction rule attached to each rule gateway R1
... Rn is the same Task as the incoming Task of the Inclusive gateway, which
precedes the rule gateways. The triggeredEventExpr of a reaction rule attached to a
rule gateway is the same Task as the outgoing Task of the rule gateway.
context RuleGateway
inv: let InSequenceFlow : SequenceFlow.allInstances()->select(c | c.targetRef
                              = this)->asSequence()->first() in
       let OutSequenceFlow : SequenceFlow.allInstances()->select(c |
                                c.sourceRef = this)->asSequence()->first() in
        rule->first().triggeredEventExpr.task = OutSequenceFlow.targetRef and
        rule->first().triggeringEventExpr.task =
                                              InSequenceFlow.sourceRef.sourceRef
```

4.3.2.2. The “Structured Synchronizing Merge” Pattern

This pattern represents convergence of two or more branches into a single subsequent branch. The pattern provides a means of merging of branches resulting from a specific Multi-Choice pattern earlier in a workflow into a single branch. Implicit action in this merging is the synchronization of all the threads of execution resulting from the preceding Multi-Choice pattern. In Figure 177, we have actually a Multi-Choice pattern (discussed in the previous subsection), where activity of one of the types *activity2 ... activityn+1* or any combination of them in parallel may thus be performed, but with an exception that we have here a synchronization of all the branches into one sequence flow by using an inclusive gateway. The challenge with this pattern is that we do not know how many tokens (generated from a Multi-Choice pattern earlier) will be arriving for synchronization. Thus, we must be able to determine how many tokens are generated upstream, and for this we can use a reaction rule attached to the rule gateway (R_m) in Figure 177, by using the shared vocabulary.

This pattern is only partially supported in BPMN with the OR gateway, because this pattern assumes a structured workflow context [148].

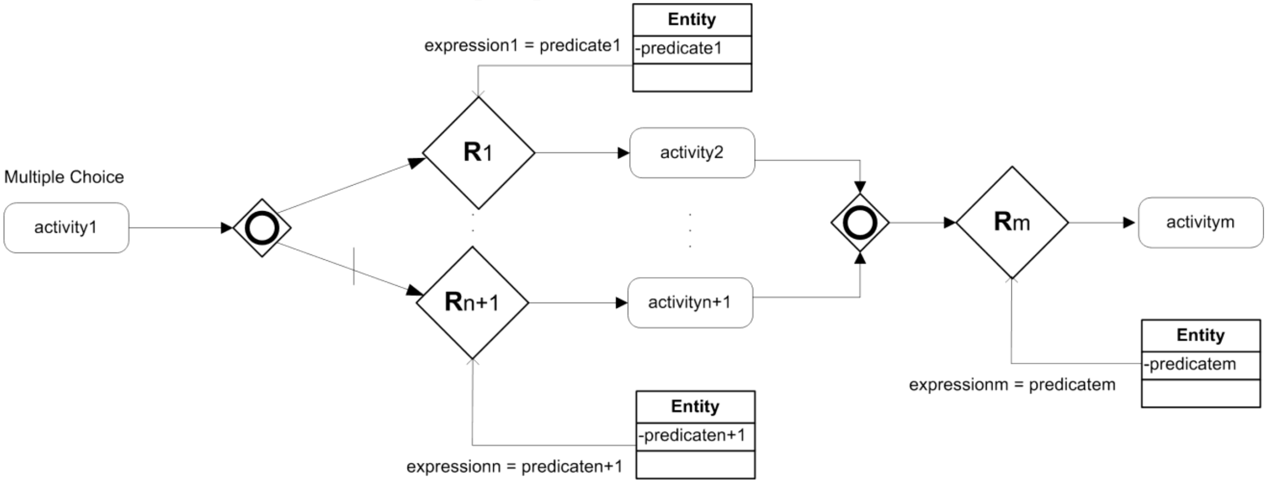


Figure 177. The “Structured Synchronizing Merge” pattern

The OCL constraints for this pattern are similar as for the previous pattern.

4.3.2.3. The “Multiple Merge” Pattern

The pattern “Multiple Merge” starts an activity once for each time two or more preceding activities end. It models convergence of two or more branches into a single branch. In Figure 178, an activity of the type *activityn+1* is started when any preceding activity out of possibly parallel activities of the type’s *activity2 … activityn* ends. An activity of the type *activityn+1* thus is performed as many times as the number of the preceding activities. The reaction rule attached to the rule gateway is used to choose two or more activities. As this pattern is very similar to the “Simple Merge” pattern, the same OCL constraint holds here.

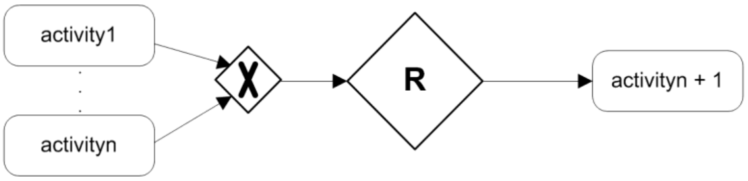


Figure 178. The “Multiple Merge” pattern

4.3.2.4. The “Discriminator” Pattern

The pattern “Discriminator” models a point in a business process that waits for one of the preceding, possibly parallel activities to complete before starting the subsequent activity. From that moment on, it waits for all remaining preceding activities to complete and “ignores” them. Once all preceding activities have been completed, it “resets” itself, so that it can be started again. The pattern “Discriminator” generalizes to the pattern “N-out-of-M-join” where N threads from M incoming transitions are synchronized. This generalized pattern “Discriminator” can be modeled as a reaction rule attached to the rule gateway with a counter counting the number of the rule’s triggering events of the activity type and that have occurred. A model of this pattern is shown in Figure 179. The value of the counter variable is increased when every preceding activity (*activity1 ... actitvityn*) ends. The precondition of this pattern is based on the two variables: the *counter* variable set to 0 (as the input parameter) and the *count* variable set to the number of the wanted preceding activities.

We supported this pattern by using a rule gateway. Without the rule gateway, this pattern in the standard BPMN can partially be supported, because this pattern can be supported only in the case of a multiple instance task, and because it is not clear how the *IncomingCondition* expression on the COMPLEX-join gateway is defined [148].

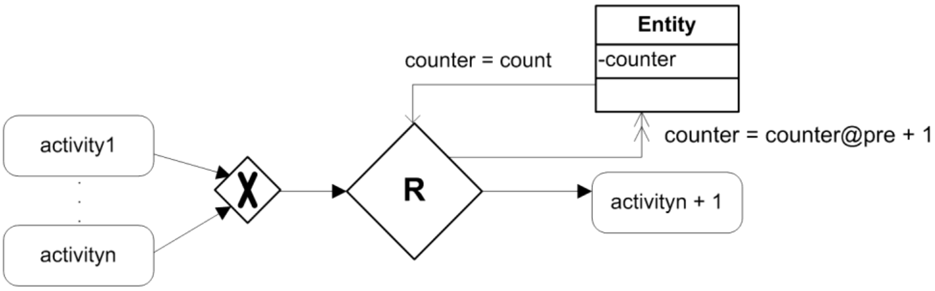


Figure 179. The "Discriminator" pattern

For this pattern, we have defined following OCL constraints:

[1] The reaction rule attached to the rule gateway, for its *triggeringEventExpr*, must have an *ChoiceEventExpression*; for its *triggeredEventExpr* must have a *Task*; and for its *postcondition* must have a *DatatypePredicateAtom.*

```
context RuleGateway
inv: rule->first().triggeredEventExpr.oclIsKindOf(Task) and
      rule->first().triggeringEventExpr.oclIsKindOf(ChoiceEventExpression) and
       rule->first().postcondition.oclIsKindOf(DatatypePredicateAtom)
```

[2] The reaction rule attached to the rule gateway, for its *triggeredEventExpr* must have a *Task*, which is the same as the outgoing *Task* of the rule gateway. This rule also must have the same *Tasks* in its *ChoiceEventExpression*, as those incoming *Tasks* of the rule gateway.

```
context RuleGateway
inv: let InSequenceFlow : SequenceFlow.allInstances()->select(c | c.targetRef
                              = this)->asSequence()->first() in
       let OutSequenceFlow : SequenceFlow.allInstances()->select(c |
                                c.sourceRef = this)->asSequence()->first() in
         let InActivites : SequenceFlow.allInstances->select(c | c.targetRef =
                                  InSequenceFlow.sourceRef)->asSequence()
                                            ->collect(e | e.sourceRef) in
        rule->first().triggeredEventExpr.task = OutSequenceFlow.targetRef and
        rule->first().triggeringEventExpr.task->forAll(p | InActivities->exists(p))
```

4.3.3. Structural Patterns

Patterns in this group characterize design restrictions that specific workflow languages may have on the form of a process model that they are able to represent and how these models behave at runtime. They include two main aspects: loops and termination of a process instance.

4.3.3.1. The "Arbitrary Cycles" Pattern

The Arbitrary Cycle pattern is a mechanism for allowing sections of a process to be repeated (as a loop). This pattern allows looping that is unstructured or not block structured. The looping part of the process may allow more than one entry or exit point. This pattern is important for the visualization of valid, but complex, looping situations in a single diagram. An rBPMN model of this pattern is shown in Figure 180. This pattern in rBPMN is represented by two reaction rules attached to two rule gateways: the rule gateway (R_1) used to split the sequence flow and the rule gateway (R_2) used to handle the loop exit condition in a process. The rule gateway (R_2) based on its condition decides whether to return the sequence flow to the *activity3* (loop) or to *activity5* (end loop).

This pattern can be supported in the standard BPMN by using Exclusive gateways, but the advantage of our solution is that we can more precisely define entry and exit looping conditions, as well as those conditions could be changed at both run- and design-time. We should also note that instead of a reaction rule attached to the rule gateway (R_1), we could use a production rule also, which is invoked on some predefined condition. Based on this condition, the whole loop process can begin as well without a need to have a triggering event.

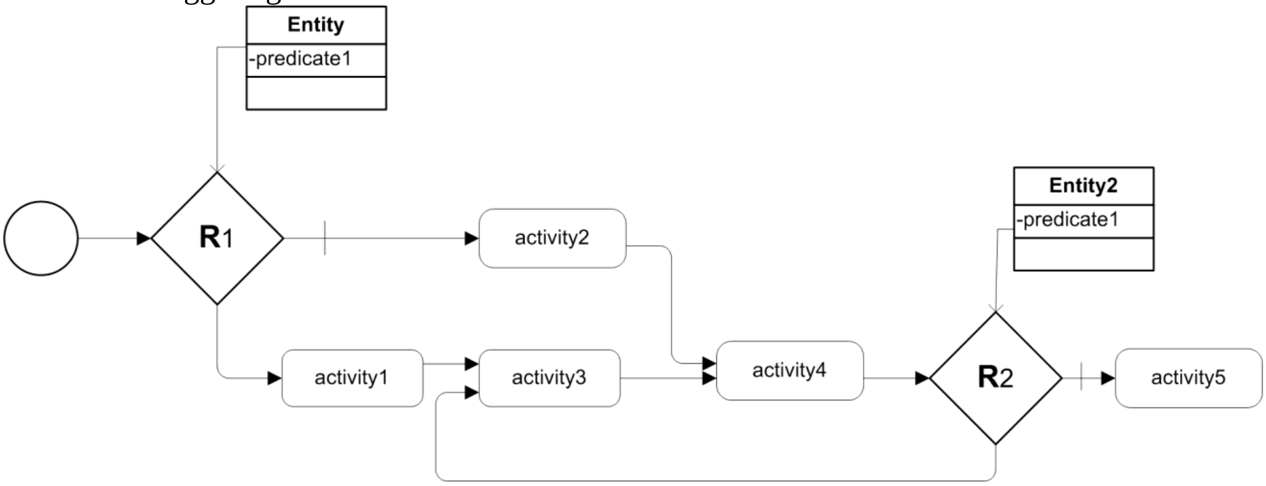


Figure 180. The “Arbitrary Cycles” pattern

The OCL constraints for this pattern are similar to the constraints in the “Exclusive Choice” pattern.

4.3.3.2. The “Implicit Termination” Pattern

The pattern “Implicit Termination” specifies that a given sub-process should be terminated when there are no remaining activities to be completed in the business process and no other activity can be started. In rBPMN, we supported this pattern through an implicit termination of an activity when all its sub-activities have completed. An rBPMN model of this pattern is shown in Figure 181. We use the end event to show that a particular path has completed, just as we would do in the standard BPMN [148]. However, we used reaction rules attached to the rule gateways after *activity2* and *activity3* tasks, in both paths, in order to dynamically determine whether the path should end or not. By combining these two rules, we can also make a ordering, when each of the sequences will end first.

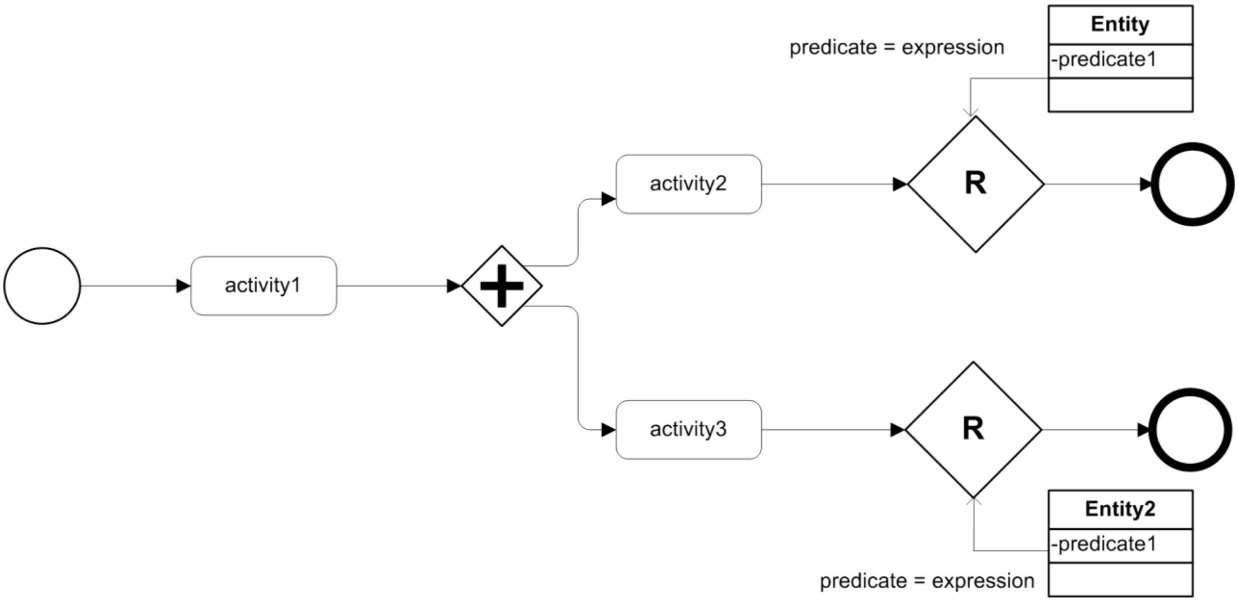


Figure 181. The “Implicit Termination” pattern

For this pattern, we have defined following OCL constraints:

[1] The reaction rule attached to the rule gateway for its *triggeringEventExpr* must have an *R2MLTriggeringTask*, while for its *triggeredEventExpr* must have an *EndEvent*.

```
context RuleGateway
inv: rule->first().triggeredEventExpr.event.oclIsKindOf(EndEvent) and
      rule->first().triggeringEventExpr.task.oclIsKindOf(R2MLTriggeringTask)
```

[2] The reaction rule attached to the rule gateway needs to have the same Event for its *triggeredEventExpr*, as the rule gateway outgoing Event, and the same Task for its *triggeringEventExpr*, as the rule gateway incoming Task.

```
context RuleGateway
inv: let InSequenceFlow : SequenceFlow.allInstances()->select(c | c.targetRef
                              = this)->asSequence()->first() in
          let OutSequenceFlow : SequenceFlow.allInstances()->select(c |
                                  c.sourceRef = this)->asSequence()->first() in
             rule->first().triggeringEventExpr.task = InSequenceFlow.sourceRef
                     and rule->first().triggeredEventExpr.event =
                              OutSequenceFlow.targetRef
```

4.3.3.3. The “N out of M join” Pattern

As mentioned in the “Discriminator” pattern, this pattern also merges many execution paths. It performs a partial synchronization and executes the subsequent activity only once. In this case, instead of waiting for all preceding activates to be received, the N out of M pattern allows the modeler to define how many of the incoming activities are necessary to continue. We enabled this by using the “count” input parameter for a reaction rule attached to the rule gateway and shown in the “Discriminator” patterns (see section 4.3.2.4).

4.3.4. Multiple Instance Patterns

These patterns are used in a process where there are multiple instances of an activity active at the same time for the same process instance. Multiple instances can arise when an activity is able to create multiple instances of it once the activity is triggered. That is, a given activity is initiated multiple times as a consequence of receiving several independent triggers, or when two or more activities in the process share the same implementation definition.

4.3.4.1. The “Multiple Instances without Synchronization” pattern

In a given process instance, multiple instances of an activity can be created. These instances are independent of each other and run concurrently. All the instances of the activity will be completed before the process continues. This means that only one Token will continue through the Process. This pattern, as well as other patterns in this group, can be model in two ways in rBPMN. In the first way by using the standard BPMN, to enable a task to have multiple instances, we need to ensure that the task’s *LoopType* attribute is set to the “MultiInstance” and the *MI_FlowCondition* is set to “None” to ensure that the process continues after the activity task instance execution [148]. However, the problem with this approach is that we cannot control how many instances are created, nor to dynamically change the number of instances. Therefore, we propose a solution in rBPMN shown in Figure 182. Here, we employ a reaction rules attached to the rule gateway to precisely determine (based on “count” attribute number) how many instances of the *activity1* will be created. We use two reaction rules attached to the rule gateway, one with positive and one with negative the same condition. This is the case as reaction rule can have only one condition defined on itself. The subsequent activity (*activity2*) is initiated after the execution of the “*activity1*” instance, if the rules’ condition is satisfied (counter=count). If not, the *MI_Condition* attribute of the *activity1* is set to 1, so that only one instance of this activity is created in each iteration. After each iteration, the reaction rule updates (denoted with *U* and double-headed arrow) *Entity*’s *counter* attribute, by adding 1 to its previous value.

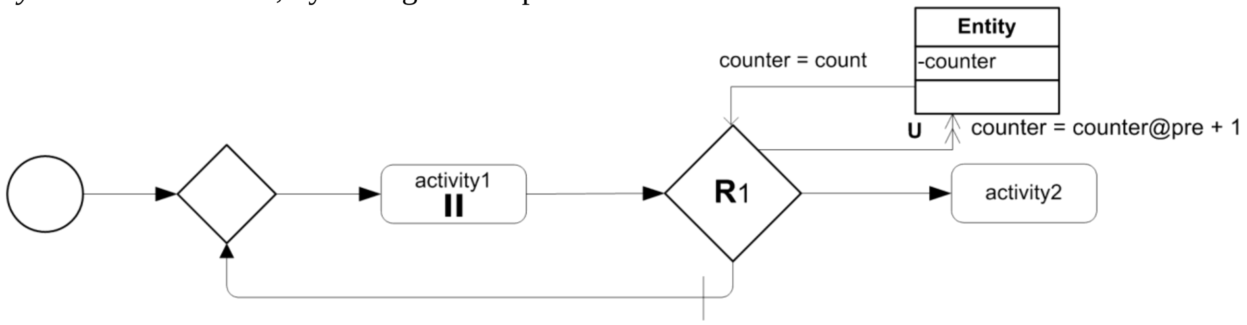


Figure 182. The “Multiple Instances without Synchronization” pattern

For this pattern, we have defined following OCL constraints:

```
[1] The rule gateway must have two reaction rules attached to it, as we need to
have one rule with positive and one with negative condition.
context RuleGateway
inv: rule->count() = 2

[2] The reaction rule attached to the rule gateway with the negated condition has
for its triggeringEventExpr an R2MLTriggeringTask, and for its triggeredEventExpr a
R2MLTriggeredGateway.
context RuleGateway
inv:      rule->select(c      |      c.condition->first().isNegated)      implies
triggeringEventExpr.oclIsKindOf(R2MLTriggeringTask)                           and
triggeredEventExpr.oclIsKindOf(R2MLTriggeredGateway)

[2] The reaction rule attached to the rule gateway with non-negated condition has
for its triggeringEventExpr an R2MLTriggeringTask, and for its triggeredEventExpr a
R2MLTriggeredTask.
context RuleGateway
```

```
inv:    rule->select(c    |    not    c.condition->first().isNegated)    implies
triggeringEventExpr.oclIsKindOf(R2MLTriggeringTask)                         and
triggeredEventExpr.oclIsKindOf(R2MLTriggeredTask)
```

[3] The reaction rule, whose condition is not negated, must have the same *Task* for its *triggeringEventExpr* as the rule gateway's incoming Task, and for its *triggeredEventExpr* must have the same *Task* as the rule gateway's outgoing (default) *Task*. In addition, the reaction rule with the negated condition, must have the same *Task* for its *triggeringEventExpr* as the rule gateway's incoming Task, and for its *triggeredEventExpr* must have the same *Gateway* as the rule gateway's outgoing (non-default) *Gateway.*

```
context RuleGateway
inv: let InSequenceFlow : SequenceFlow.allInstances()->select(c | c.targetRef
                              = this)->asSequence()->first() in
       let OutSequenceFlowNonDefault : SequenceFlow.allInstances()->select(c |
                                c.sourceRef = this)->select(e | e.
                                    ConditionType <> "Default")->first() in
         let OutSequenceFlowDefault : SequenceFlow.allInstances()->select(c |
                                c.sourceRef = this)->select(e | e.
                                    ConditionType = "Default")->first() in
           let NegatedRule : rule->select (c | c.conditions->
                                       first().isNegated = "true")->first() in
              let NonNegatedRule : rule->select (c | c.conditions->
                                      first().isNegated = "false")->first() in



     NegatedRule.triggeringEventExpr.task = InSequenceFlow.sourceRef
      and
     NonNegatedRule.triggeringEventExpr.task = InSequenceFlow.sourceRef
      and
  NegatedRule.triggeredEventExpr.gateway = OutSequenceFlowNonDefault.targetRef
      and
     NonNegatedRule.triggeredEventExpr.task = OutSequenceFlowDefault.targetRef
```

4.3.4.2. The "Multiple Instances with a Priori Known Design Time Knowledge" pattern

The pattern "Multiple Instances with a Priori Known Design Time Knowledge" is similar to the previous "Multiple Instances without Synchronization" pattern. This enables creating many instances of one activity. The number of instances is known at design time. However, in this case, all instances of the multi-instance task must be completed before the subsequent activity is initiated (i.e. the multiple instances must be synchronized). This pattern can be implemented by defining a multi-instance task, which has the *MI_FlowCondition* attribute set to "All", so that the process flow continues when all instances of an activity end [148]. However, the problem with this solution is to define precisely the number of created instances and the condition when those instances have completed.

An rBPMN model is shown in Figure 183, where the *activity2* is replicated a fixed number times (i.e., two times in our example) to be executed in parallel or sequentially. This number is defined at design time by using the condition on the reaction rule attached to the rule gateway. The *LoopType* attribute of the *activity2* is set to MultiInstance. The *activity2* task is to synchronize multiple instances of the Subprocess shown in Figure 183, so that once all sub activities in the Subprocess end, the *activity2* also ends, and the next *activity3* is started (or the business process ends).

In this pattern, we have two reaction rules attached to the rule gateway. The first rule is invoked when input condition (*counter=2*) is false. In this case, the crossed sequence flow from the rule gateway is chosen (the *activity2* MI task is invoked). The second rule is invoked when the input condition (*counter=2*) is true. When this happens the Subprocess ends, and the *activity3* task is invoked.

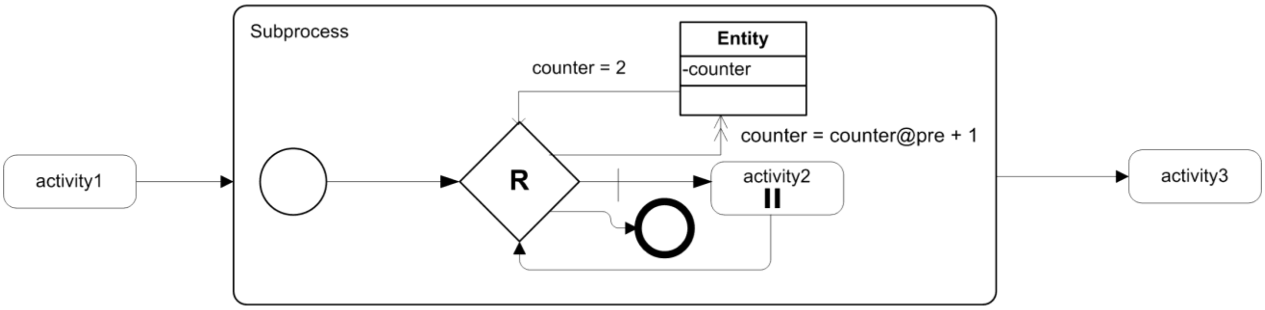


Figure 183. The "Multiple Instances with a Priori Known Design Time Knowledge" pattern

For this pattern, we have defined following OCL constraints:

[1] The rule gateway has two reaction rules attached to it.

```
context RuleGateway
inv: rule->count() = 2
```

[2] The reaction rule attached to the rule gateway with the negated condition has, for its *triggeringEventExpr*, an *R2MLTriggeringEvent*, and for its *triggeredEventExpr* an *R2MLTriggeredEvent*.

```
context RuleGateway
inv: rule->select(c | c.condition->first().isNegated) implies
triggeringEventExpr.oclIsKindOf(R2MLTriggeringEvent) and
triggeredEventExpr.oclIsKindOf(R2MLTriggeredEvent)
```

[3] The reaction rule attached to the rule gateway with non-negated condition has, for its *triggeringEventExpr*, an *R2MLTriggeringEvent*, and for its *triggeredEventExpr* a *R2MLTriggeredTask*.

```
context RuleGateway
inv: rule->select(c | not c.condition->first().isNegated) implies
triggeringEventExpr.oclIsKindOf(R2MLTriggeringEvent) and
triggeredEventExpr.oclIsKindOf(R2MLTriggeredTask)
```

[4] The reaction rule, whose condition is not negated, must have the same *Event* for its *triggeringEventExpr* as the rule gateway's incoming Event, and for its *triggeredEventExpr* must have the same *Task* as the rule gateway's outgoing (non-default) *Task*. In addition, the reaction rule with the negated condition, must have the same *Event*, for its *triggeringEventExpr*, as the rule gateway's incoming Event, and for its *triggeredEventExpr* must have the same *Event* as the rule gateway's outgoing *Event.*

```
context RuleGateway
inv: let InSequenceFlow : SequenceFlow.allInstances()->select(c | c.targetRef
                              = this)->asSequence()->first() in
       let OutSequenceFlowNonDefault : SequenceFlow.allInstances()->select(c |
                                c.sourceRef = this)->select(e | e.
                                    ConditionType <> "Default")->first() in
         let OutSequenceFlowDefault : SequenceFlow.allInstances()->select(c |
                                c.sourceRef = this)->select(e | e.
                                    ConditionType = "Default")->first() in
           let NegatedRule : rule->select (c | c.conditions->
                                       first().isNegated = "true")->first() in
              let NonNegatedRule : rule->select (c | c.conditions->
                                      first().isNegated = "false")->first() in
```

```
NegatedRule.triggeringEventExpr.event = InSequenceFlow.sourceRef
  and
NonNegatedRule.triggeringEventExpr.event = InSequenceFlow.sourceRef
  and
NegatedRule.triggeredEventExpr.event = OutSequenceFlowNonDefault.targetRef
  and
NonNegatedRule.triggeredEventExpr.task = OutSequenceFlowDefault.targetRef
```

4.3.4.3. The "Multiple Instances with a Priori Known Runtime Knowledge" Pattern

In the pattern "Multiple Instances with a Priori Known Runtime Knowledge", the number of instances of a given activity in a process is variable and may depend on characteristics of the business process instance or availability of resources. However, the number of instances is known at some stage during run-time, before the instances of that activity type need to be created. Once all instances are completed, an activity of some other type needs to be started. In Figure 184, the looping subprocess includes an *activity2* which is started after the end of an *activity1*, and enabled by a reaction rule attached to the rule gateway. As in the previous pattern, we also have here two reaction rules attached to the rule gateway, but in this case, the condition is *counter=count*. Upon the start of a Subprocess, the *activity2* task is performed for each its instance as many times as it is defined by a *count* variable. This variable may be set during the process by some task or by using an R2ML reaction or production rule. The activities of the type *activity2* are executed sequentially. In the model of the pattern depicted in Figure 184, an implicit termination of the Subprocess is when all its sub-activities are completed and the *Entity*'s "counter" attribute is equal to the "count" variable. The "counter" variable is pre-set to 0. In this case, the reaction rule evaluates to false, and the Subprocess ends after end of all *activity2* instances.

In the standard BPMN, this pattern is supported by setting a multiple instance task's MI_Condition attribute to the number of required instances at "runtime" [148]. However, the solution with rBPMN is more flexible, as we can define more complex conditions on a rule gateway that control the execution of the *activity2* in the Subprocess. We should also mention that the standard BPMN [88] does not have an integrated language for defining such expressions (conditions).

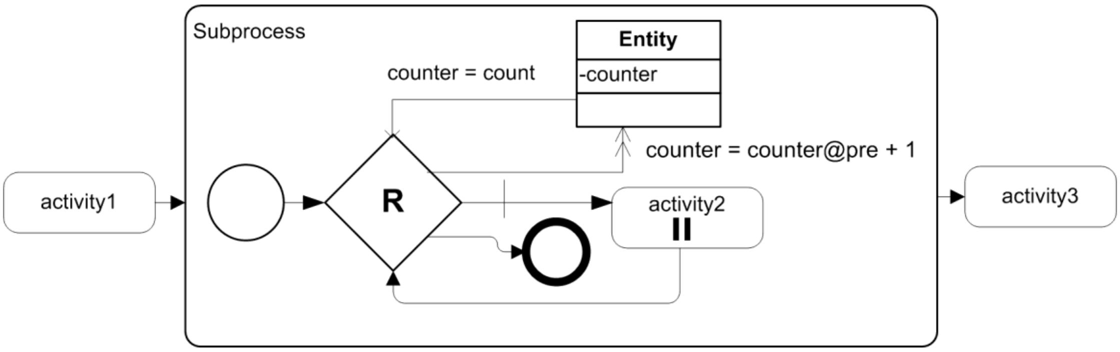


Figure 184. The "Multiple Instances with a Priori Known Runtime Knowledge" pattern

The OCL constraints for this pattern are the same as for previous pattern.

4.3.4.4. The "Multiple Instances with no a Priori Runtime Knowledge" Pattern

In this pattern, the number of instances of a given activity is not known at design-time, nor it is known at any stage during runtime, until immediately before the instances of that activity type need to be created. Once all (multiple-)instances are completed, an activity of some other type needs to be started. The difference from the pattern "Multiple Instances with a Priori Runtime Knowledge" is that even while some of the activity instances are being executed or have already completed new ones can be created. An rBPMN model of this pattern is shown in Figure 185. The Subprocess is started after the end of *activity1*. In this Subprocess, *activity3* is invoked multiple times, and *activity2* along with the reaction rule attached to the rule gateway is used to determine if more instances of *activity3* are needed. The *activity2* can also increase an *Entity's* attribute *count*, so that the number of instances increases. Upon the start of the Subprocess, if the condition attached to the rule gateway (and its associated reaction rules, as in the previous pattern) *counter = count* is false, the reaction rule with non-negated condition passes to the parallel gateway that executes activity2 and activity3 instances in parallel. The *Entity*'s *counter* attribute is pre set to 0 before entering the Subprocess. The *activity2* checks if more instances of *activity3* are needed and record the decision by increasing the *count* attribute of the type *Entity*. The completion of *activity2* again invokes the rule gateway. This loop continues until all instances of *activity3* are completed. Similar to the pattern "Multiple Instances with a Priori Runtime Knowledge," synchronization of multiple instances of *activity3* is achieved through the implicit termination of the Subprocess when all its instances have completed. This actually means that when all *activity3* instances have ended, the Subprocess also ends, and the next *activity4* is started (or the business process ends).

By employing reaction rules in this pattern, we supported this pattern, which is not supported in the standard BPMN as it lacks features to create new task instances dynamically [148].

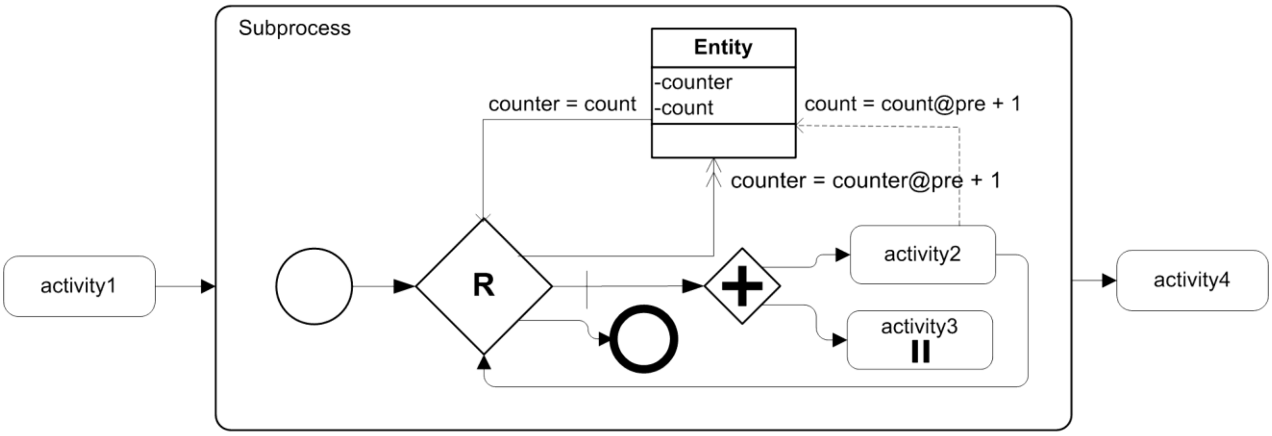


Figure 185. The "Multiple Instances with no a Priori Runtime Knowledge" pattern

For this pattern, we have the same OCL constraints, as for the "Multiple Instances with a Priori Known Design Time Knowledge" pattern. An exception is that for *triggeredEventExpr* of both reaction rules we have *ParallelEventExpression* with two *Tasks* (*activity2* and *activity3*).

[1] The reaction rule with a non-negated condition must have the same *Event* for its *triggeringEventExpr* as the rule gateway's incoming Event, and the same parallel tasks for its *triggeredEventExpr*, as the rule gateway's outgoing tasks preceded by a parallel gateway. In addition, the reaction rule with the negated condition must have the same *Event* for its *triggeringEventExpr* as the rule gateway's incoming Event, and for its *triggeredEventExpr* must have the same *Event* as the rule gateway's outgoing *Event.*

```
context RuleGateway
inv: let InSequenceFlow : SequenceFlow.allInstances()->select(c | c.targetRef
                              = this)->asSequence()->first() in
       let OutSequenceFlowNonDefault : SequenceFlow.allInstances()->select(c |
                                c.sourceRef = this)->select(e | e.
                                    ConditionType <> "Default")->first() in
         let OutSequenceFlowDefault : SequenceFlow.allInstances()->select(c |
                                c.sourceRef = this)->select(e | e.
                                    ConditionType = "Default")->first() in
            let NegatedRule : rule->select (c | c.conditions->
                                       first().isNegated = "true")->first() in
            let NonNegatedRule : rule->select (c | c.conditions->
                                      first().isNegated = "false")->first() in
             let OutSequenceFlowParallelFlows : SequenceFlow.allInstances()->
                 select(c | c.sourceRef = OutSequenceFlowDefault.targetRef) in

    NegatedRule.triggeringEventExpr.event = InSequenceFlow.sourceRef
      and
    NonNegatedRule.triggeringEventExpr.event = InSequenceFlow.sourceRef
      and
    NegatedRule.triggeredEventExpr.event = OutSequenceFlowNonDefault.targetRef
      and
   SequenceFlow.allInstances()->select(c | c.sourceRef =
               NonNegatedRule.triggeredEventExpr.gateway)->asSequence()
                         ->forAll(e | OutSequenceFlowParallelFlows->exists(e))
```

4.3.5. State-based Patterns

These patterns define situations for which solutions are most easily accomplished in process languages that support the notion of state. A state of a process instance includes a broad collection of data associated with the current execution including the status of various activities. The patterns in this group reflect the possibility that the business processes could be affected by factors outside of the business process engine.

4.3.5.1. The "Deferred Choice" Pattern

The "Deferred Choice" pattern is a divergence point in a business process where one of several possible branches is chosen. In this pattern, an environment selects an activity to be performed, usually by employing an event. Only one of the alternative activities is executed. This means that once the environment triggers one of the activities, the other alternative activities are withdrawn. According to the example of the "Deferred Choice" pattern in Figure 186, after the end of *activity1*, by using the Event-based exclusive gateway, only one of activities *activity2* or *activity3* is started depending on which intermediate message is *first received*. The rule gateways are used in both cases to decide whether the sequence flow should be followed or not, based on the predefined condition of the reaction rules attached to the rule gateways. So, the choice is made by the environment (event), and we additionally constrained every decision by using reaction rules. In this way, we enabled more precisely definition of this pattern. In the standard BPMN without the rule gateways, it is not possible to dynamically determine under what conditions activities should be invoked [148].

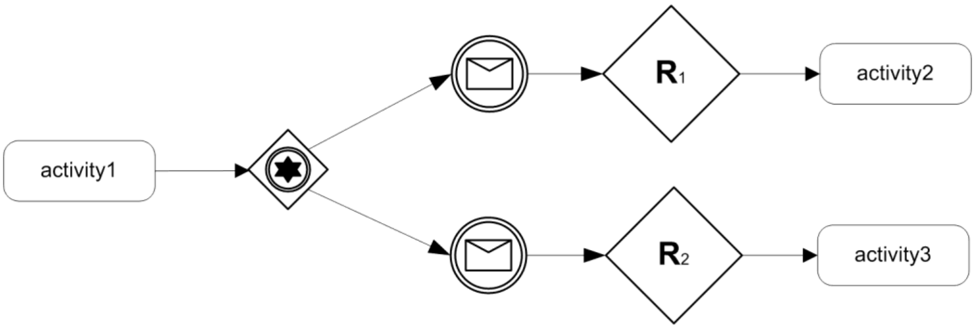


Figure 186. The “Deferred Choice” pattern

For this pattern, we have defined following OCL constraints:

```
[1] A reaction rule attached to a rule gateway has for its triggeringEventExpr an
Event, and for its triggeredEventExpr a Task.
context RuleGateway
inv: rule->first().triggeringEventExpr.event.oclIsKindOf(Event) and
     rule->first().triggeredEventExpr.task.oclIsKindOf(Task)

[2] A reaction rule attached to a rule gateway must have the same Event for its
triggeringEventExpr, as the rule gateway’s incoming Event, and the same Task for
its triggeredEventExpr, as the rule gateway’s outgoing Task.

context RuleGateway
inv: let InSequenceFlow : SequenceFlow.allInstances()->select(c | c.targetRef
                              = this)->asSequence()->first() in
          let OutSequenceFlow : SequenceFlow.allInstances()->select(c |
                                   c.sourceRef = this)->asSequence()->first() in
             rule->first().triggeredEventExpr.task = OutSequenceFlow.targetRef
                     and rule->first().triggeringEventExpr.event =
                            InSequenceFlow.sourceRef
```

4.3.5.2. The “Milestone” Pattern

The “Milestone” pattern enables an activity until a milestone (specific state) is reached. When the milestone is reached, the nominated task can be enabled. In this pattern, only one activity is selected to be performed, while all activities are completed and the end of the process is reached. According to the example of the pattern “Milestone” in Figure 187, after the start of a process, the sequence flow splits by using the parallel gateway. The first branch invokes *activity1* and the flow reaches the rule gateway (R_1), which enables the *Entitys*’ “enabled” attribute. The second branch invokes *activity2* and reaches the rule gateway (R_2), which then use the *Entity*’s “enabled” attribute for a condition to decide which flow will be selected. If the “enabled” attribute is *true*, then *activity3* is invoked and the sequence flow is returned to the start of the branch. If the “enabled” attribute is *false*, *activity4* is selected and the second branch ends. After invocation of the rule gateway (R_1) in the first branch, *activity5* is invoked, and the rule gateway (R_3) is reached. The rule gateway (R_3) then sets *Entity*’s “enabled” attribute to *false*, and the first branch ends. When this happens, *activity3* in the second branch cannot be invoked anymore, because the condition *enabled=true* on the rule gateway (R_2) is not true. Each of the three rule gateways has a reaction rule attached to them.

This pattern is not supported in the standard BPMN due to the lack of modeling support for states [148], which we solved by using reaction rules in rBPMN.

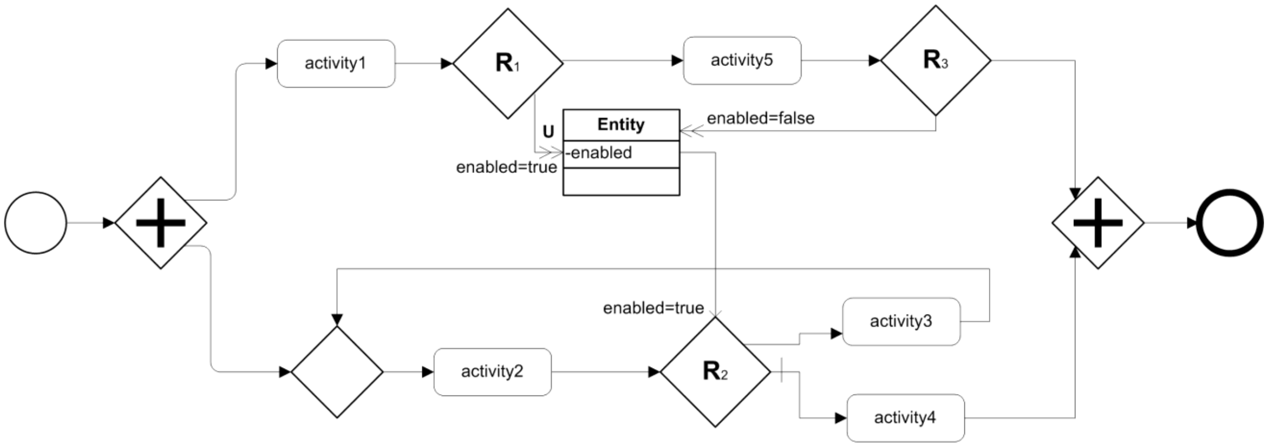


Figure 187. The “Milestone” pattern

4.3.6. Cancellation Patterns

These patterns characterize the concept of activity or process cancellation where enabled or active activity instances are withdrawn. Various forms of exception handling in process are also based on cancellation concepts.

4.3.6.1. The “Cancel Activity” Pattern

This pattern defines the ability that an enabled activity can be cancelled in some situations. If a task is started, it is stopped and the sequence flow is redirected. We show this pattern in Figure 188, where the exception handling is executed through an intermediate exception event that is attached to the boundary of *activity1*. If the trigger for the intermediate event occurs while *activity1* is being performed, *activity1* will be interrupted and the sequence flow will proceed through the intermediate event and proceed down the sequence flow to the rule gateway and to the *Exception handler* subprocess. This pattern is supported similarly in the standard BPMN [148], but we used the rule gateway to define the condition on the exception flow. This use of rule gateways can define whether the *Exception handler* will be invoked or not. The *Exception handler* subprocess is responsible to handle the generated exception, where in this case we just pass the message to the end event.

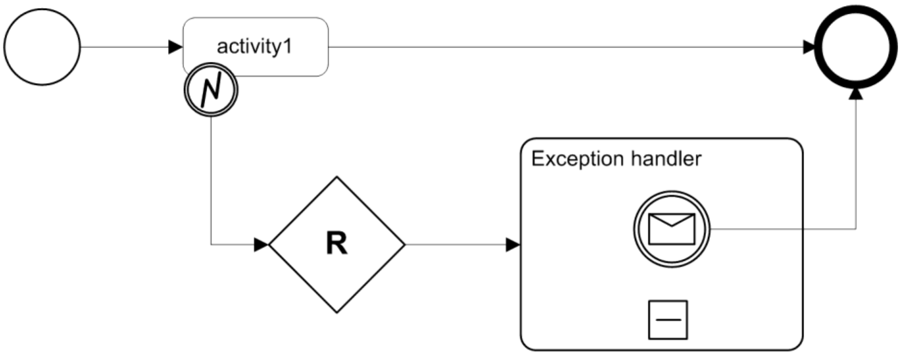


Figure 188. The “Cancel Activity” pattern

For this pattern, we have defined following OCL constraints:

```
[1] The reaction rule attached to the rule gateway has for its triggeringEventExpr
an Event, and for its triggeredEventExpr a SubProcess.
context RuleGateway
inv: rule->first().triggeringEventExpr.event.oclIsKindOf(Event) and
     rule->first().triggeredEventExpr.subProcess.oclIsKindOf(SubProcess)
```

```
[2] The reaction rule attached to the rule gateway need to have the same Event for
its triggeringEventExpr, as the rule gateways' incoming Event, and the same Task
for its triggeredEventExpr, as the rule gateways' outgoing SubProcess.

context RuleGateway
inv: let InSequenceFlow : SequenceFlow.allInstances()->select(c | c.targetRef
                              = this)->asSequence()->first() in
          let OutSequenceFlow : SequenceFlow.allInstances()->select(c |
                                 c.sourceRef = this)->asSequence()->first() in
             rule->first().triggeredEventExpr.subProcess =
                  OutSequenceFlow.targetRef and
                  rule->first().triggeringEventExpr.event =
                                          InSequenceFlow.sourceRef
```

4.3.6.2. The "Cancel Case" Pattern

The pattern "Cancel Case" in Figure 189 cancels the whole business process instance being executed. This pattern is similar to the Cancel Activity pattern. In this situation, however, an intermediate event is attached to the boundary of a Subprocess that contains other activities, rather than an activity (see Figure 189). In this pattern, we have a Subprocess that is initiated by a start event and followed by a rule gateway with two reaction rules attached to it. The reaction rules attached to the rule gateway then decides based on their condition, whether to invoke *activity1* or to cancel the *Subprocess*. The *Subprocess* can also be cancelled in the case if the *activity1* task generates and error during its execution. When the exception is generated, the *Fault handler* attached to the *Subprocess* boundary passes the sequence flow to the *Exception handler*, which is then responsible to handle the exception. In this simple case, the *Exception handler* is used just to send the message.

This pattern is supported in plain BPMN [148], however the rule gateway enabled us to cancel case not only in the case when *activity1* fails, but also if the predefined rule is not satisfied.

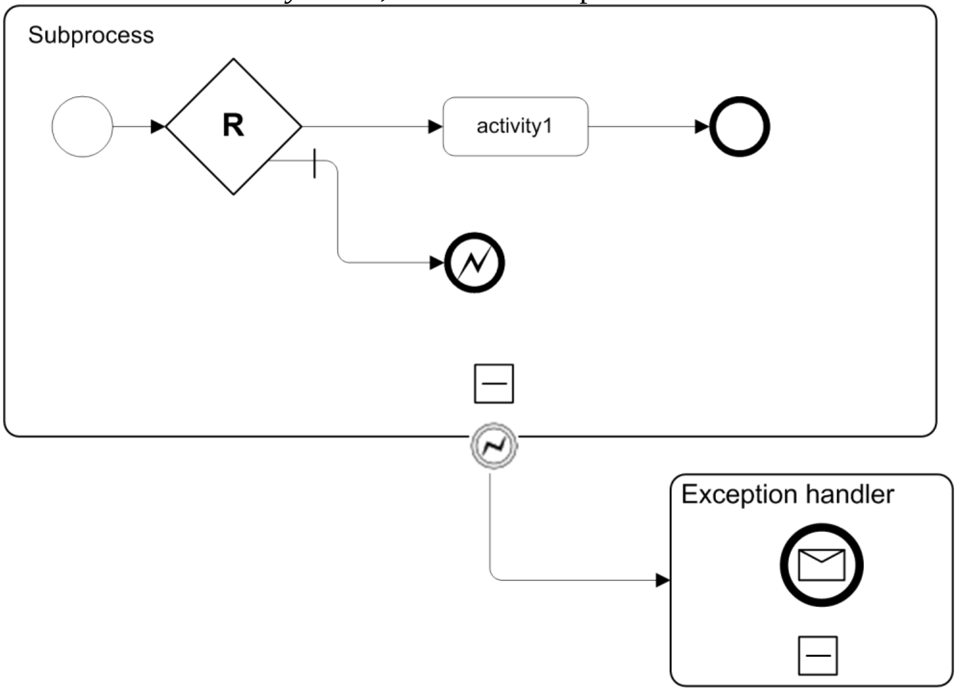


Figure 189. The "Cancel Case" pattern

For this pattern, we have defined following OCL constraints:

```
[1] The rule gateway must have exactly two reaction rules attached to it.
```

```
context RuleGateway
inv: rule->size() = 2
```

[2] The one of the outgoing sequence flows must have the "*Default*" value for its *ConditionType*.

```
context RuleGateway
inv: let OutSequenceFlow : SequenceFlow.allInstances()->select(c |
                 c.sourceRef = this)->asSequence()->first() in
                      OutSequenceFlow->exists(e | e. ConditionType = "Default")
```

[3] The reaction rule, whose condition is not negated, must have the same *Event* for its *triggeringEventExpr* as the rule gateway's incoming *Event*, and for its *triggeredEventExpr* must have the same *Task* as the rule gateway's outgoing (non-default) *Task*. In addition, the reaction rule with the negated condition, must have the same *Event* for its *triggeringEventExpr* as the rule gateway's incoming *Event*, and for its *triggeredEventExpr* must have the same *Event* as the rule gateway's outgoing (default) *Event.*

```
context RuleGateway
inv: let InSequenceFlow : SequenceFlow.allInstances()->select(c | c.targetRef
                              = this)->asSequence()->first() in
       let OutSequenceFlowNonDefault : SequenceFlow.allInstances()->select(c |
                                c.sourceRef = this)->select(e | e.
                                    ConditionType <> "Default")->first() in
         let OutSequenceFlowDefault : SequenceFlow.allInstances()->select(c |
                                c.sourceRef = this)->select(e | e.
                                    ConditionType = "Default")->first() in
            let NegatedRule : rule->select (c | c.conditions->
                                       first().isNegated = "true")->first() in
               let NonNegatedRule : rule->select (c | c.conditions->
                                      first().isNegated = "false")->first() in



  NegatedRule.triggeringEventExpr.event = InSequenceFlow.sourceRef
      and
  NonNegatedRule.triggeringEventExpr.event = InSequenceFlow.sourceRef
      and
  NegatedRule.triggeredEventExpr.event = OutSequenceFlowDefault.targetRef
      and
  NonNegatedRule.triggeredEventExpr.task = OutSequenceFlowNonDefault.targetRef
```

4.4. Business rules patterns for agile business processes

The authors of [42] [43] proposed a set of nine business process modeling patterns for full integration of business rules into business processes and how these patterns can be integrated in a standard language for SOAs (i.e., BPEL). In that research, business logics contained in business process models are externalized into business rules at design time. The assumption is that business processes can be made agile and adaptive at run-time. The externalized logic is represented by using derivation, constraint and process rules, which are related to certain parts of a business process models. Derivation rules are used for decisions in a process models, constraints are enforced by those decisions and process rules define logical dependencies of activities [43].

In section 3.1.2 we show how can rules are used in the rBPMN language, and in the subsequent subsections, we will show how these three types of rules can be used in a business process models, following the presentation order from [42]. By representing them in rBPMN, we will show how these patterns can be formalized by using a concrete rule language and make them more expressive. This will also show expressivity of rBPMN in representing these patterns, which is important for relation to SOA language, as they are mappable to BPEL language [42].

4.4.1. Control Flow Decisions

A control flow defines when different activities are executed. At a specific point in the control flow there is a possibility to enter alternate execution paths. Such points are gateways in rBPMN. The decision logic is defined by different conditions that can be assigned to each output branch. The data that is necessary to evaluate the decision logic is called input data, and the result of the evaluation is the output data (used to determine which output branch should be followed).

In this group, authors [42] [43] identified four patterns: Decision logic abstraction, Decision node to BR mapping, Decision with flexible input data and Decisions flexible output.

4.4.1.1. The "Decision Logic Abstraction" pattern

This pattern models decisions in such a way, that the decision logic is defined by a business rule (derivation) and not within the business process. The pattern allows for updating decision logic during runtime without redeploying the business process. An example of such a decision is "If a book is in the library, then the book available". In this example, the rule connects the data flow, because the rule input is a requested book, while the output is the conclusion that the book is available. Additionally, all business rules can be stored in one place, e.g., Business Rule Repository, or within a rule set.

A limitation of this pattern is the fact that input data and results of a rule are not flexible. They are defined within business processes and cannot be changed. The negative consequences are the loss of a holistic view onto a business process, because the business process decision logic is no longer visible. In rBPMN, we improve this by using URML for defining rule decision logic [42]. Business processes and business rules are developed separately, but they still use a common data model (R2ML Vocabulary). This data model can either be defined within the business process or used with business rule or the other way around.

This pattern can be modeled by inserting a rule gateway into a business process to serve as a decision node. This rule gatway has a derivation rule and two production rules attached (see Figure 190). The condition defined on the rule gateway, is mapped to the condition of the derivation rule. The condition on both production rules is the same, but only negated on the second derivation rule. Derivation rules is used to derive the fact (as described in section 3.1.2.2) and the two production rules are then used to produce the action based on that fact. If condition on the first production rule evaluates to true, the "result" sequence flow is chosen. If that condition evaluates to false, then the second production rule is invoked and the "not result" sequence flow is chosen. The data for business rules is provided by the business process (i.e., *expression* in Figure 190). Based on this decision, the business process chooses appropriate branch to continue process flow.

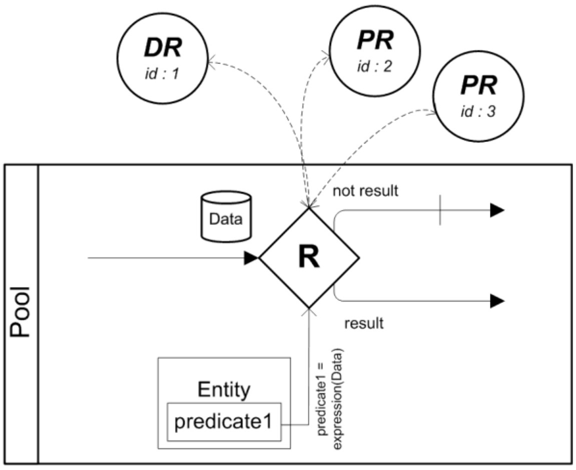


Figure 190. The “Decision logic abstraction” pattern (Boolean choice)

In the case when a decision node is not simply a boolean decision, i.e., when we have a multiple choice decision, a conditions defined on a rule gateway can be automatically serialized and the appropriate conditions can be mapped to rules attached to a rule gateway.

Here we have a rule gateway *RGj* to which we can assign *n* logical expressions: *LE = {LE1, LE2, ..., LEn}*, where for each logical expression *LEj (j=1..n)* exists exactly one outgoing sequence flow from a rule gateway. Each outgoing sequence flow (*OFj*) from a rule gateway is choosen if logical expression *LEj* is evaluated to true. However, for each logical expression, we can also have a *false* result. In that case, we need to have exactly one *else* for a rule gateway, whose logical expression is: *not (LE1 Ç LE2 Ç...Ç LEn)*. So, each logical expression on a outgoing sequence flow from a rule gateway is mapped to a R2ML rule condition: *R = {R1, R2, ..., Rn}*. This means that each rule have assigned one logical expression *LEj (j = 1..n)* for its condition. This pattern is shown in Figure 191.

In Figure 191 we have a derivation rule that is used to derive a fact (here we can have multiple derivation rules to derive more facts) and when that process is finished, all production rules are fired that have conditions satisfied (conditions based on derived facts). In addition, it is possible to have multiple branches running in parallel from a rule gateway, as a consequence of more production rules could be triggered.

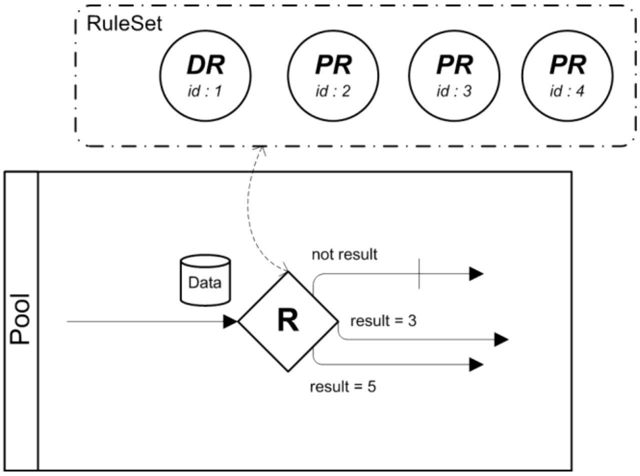


Figure 191. The “Decision logic abstraction” pattern (multiple choice)

4.4.1.2. The “Decision Node to Business Rule Binding” pattern

In this pattern, unlike the previous “Decision logic abstraction” one, a mapping is used to bind the decision node to a derivation and production rule pair dynamically. So, it is required that a business rule has a unique identifier, which is enabled in R2ML rules through their “id” property. It is important to notice here, that a rule gateway may have multiple different derivation and production rules attached to it. The condition for execution of such rule is carried out in a conclusion part of a rule. The advantage of this pattern is that it overcomes limitations of the previous one, because business rules can be reused for different decision nodes within different business processes. In addition, this pattern enables us to dynamically attach different business rules to each decision node. Other business rules can be used to define the mapping between a decision node and a rule, such as production rules. This pattern is modeled in rBPMN and is shown in Figure 192, where in the left part of the figure one possible implementation for the mapping engine is shown. The Mapping Service is responsible to choose an appropriate rule (service) to be bound to the rule gateway. In the right part of the figure, we can see a conceptual mapping between the rule gateway (i.e. decisions node) and the derivation and production rule.

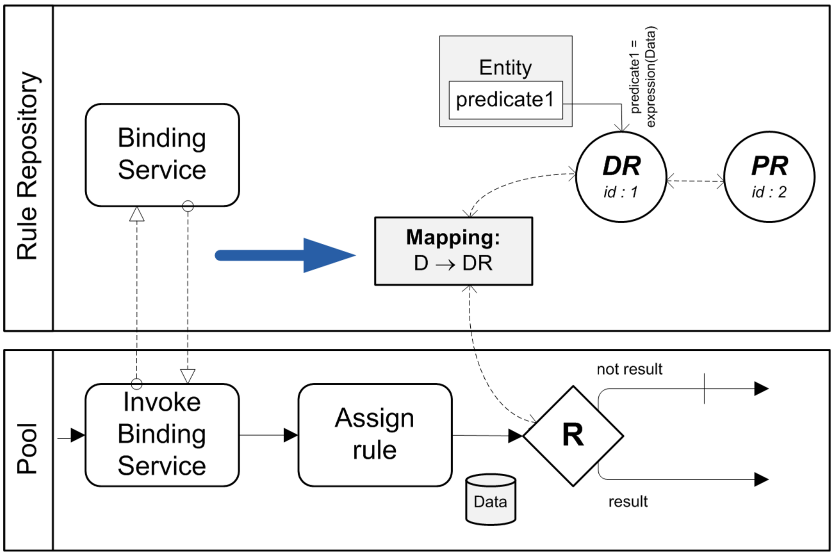


Figure 192. The "Decision node to business rule binding" pattern

4.4.1.3. The "Decision with flexible input data" pattern

This pattern differs from the previous one because the decision node calls a business rule directly without providing the normal input data, which is usually passed to a rule gateway. The business rule just gets a process identifier, to allow the business rule to access the business process context, which contains all the data that are available for the process instance. The main advantage of this pattern is that it provides the business rule with an ability to access the relevant data, instead of relying on some provided data. This allows for changing dynamically the input data for a rule during the process execution.

In this pattern, we need to populate the Process instance context object with business process instance variables, so that the rule can access them. The access to the business process instance context must be provided by a business process engine or handled within the business process. The limitation of this pattern is that not all context data of a business process are available during a business process instance runtime, so the designer of the business rule is responsible to limit the data used in a business rule to the data that are already available when the business rule is to be evaluated. Another consequence of the shared data model is that any business process schema change may make a change in the business rule necessary.

An rBPMN model of this pattern is shown in Figure 193. The data (var) which passed to a rule from a process are used to identify a business process instance. For filling the context data object (variable), we use the *FillContext* task before the rule gateway. Then, the input data from the rule attached to the rule gateway are populated from the process instance context, and the rule condition is defined by using the context variable.

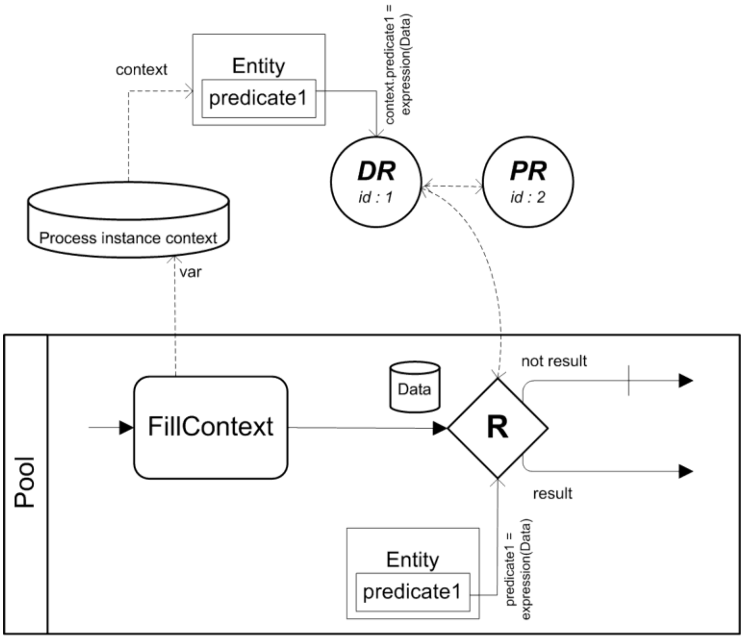


Figure 193. The "Decision with flexible input data" pattern

4.4.1.4. The "Decision flexible output" pattern

The output data of a decision node may be unknown when designing the business process. For example, the *multiple choice* decision node (gateway) may have three output branches. A rule can be used to decide if either one of the two predefined (static) branches is followed or if the third (dynamic) branch is followed. The third branch is dynamic because an activity, defined by a process rule, gets dynamically bound on to the business process flow.

This pattern enables for assigning activities dynamically to decision output branches, which makes business processes very flexible. This is especially useful for handling exceptional decision results. The limitation of this pattern is that control flow is still defined within the business process, so it is needed to be specified what happens after the execution of the dynamic activity. The output branch can be a normal decision branch, can call a decision node again, or can terminate the business process.

This pattern is shown in Figure 194. In this pattern, we have three production rules attached to the rule gateway for three output branches. If the conditions of the first two production rules are satisfied, the *result = 1* and *result = 2* flows are selected, respectively. In the case when *result = 3*, the third production rule is fired and it returns the serviceName to the *Lookup Service* task is invoked to choose a custom task to be invoked in a proceeding of a sequence flow, based on the serviceName. So, the rule gateway and its outgoing sequence flows are mapped into one of the attached rules. This enables a rule to choose a custom task, which is not initially (in design-time) defined in a standard sequence flow.

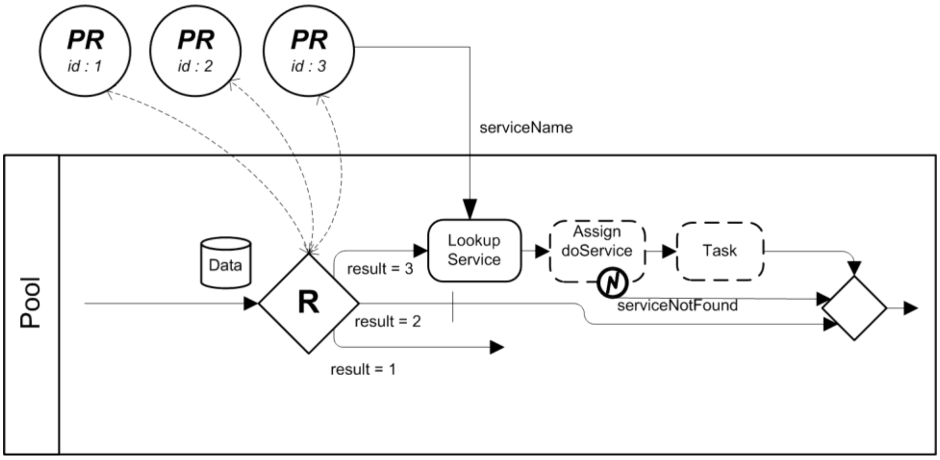


Figure 194. The “Decision flexible output” pattern

4.4.2. Data Constraints

Data constraints are used to define which values data objects can hold, and relations between them. Those constraints in process models are seen as a responsibility of the data model. The data model defines which data object can hold what kind of data. However, most of the data model defines simply data constraints, such as types of attribute values.

Nevertheless, business processes have to deal with more complex business oriented data constraints, such as “*If customer returns a car and the car has more than 5000km from the last service then send the car to the service*”. It is reported that, some of the constraints even use derived data and not business process data [42], such as “*the duration of a rental must not be greater than 3 months*”

These constraints could be modeled in a way that the data model expresses them, but this may have two major drawbacks: first, this hides the business logic from the business user, as these constraints are not seen from the business process; and second, major business process execution engines do not provide the ability to change a business process data model during runtime [42]. Additionally, constraints violation is need to be checked. Therefore, we need to check whether those constraints are satisfied or not during a process run time.

4.4.2.1. The “Constraints at predefined checkpoint” pattern

This pattern provides a solution to checking constraints validity on an activity’s input and output data. As the position of an activity in a business process is known at design-time, the activity is a predefined checkpoint for constraints. This pattern can be used in the positions in a business process control flow where constraints should get checked are known at business process modeling time, and do not change during runtime. Thus, constraints can be used to validate data before or after the activity. The usage of business rule makes it possible to update these constraints or add additional constraints during runtime.

In Figure 195, we show how this pattern is supported in rBPMN, by employing the Check constraint subprocess. This subprocess is used to check constraints for a chosen task. As data constraints in URML are represented as OCL constraints (invariants), we connect the rule gateway to the constraint (e.g., invariant, precondition, or postcondition), and if the constraint is not satisfied, an exception event is chosen from the rule gateway. Then, the Handle exception task is invoked to handle the exception. Those OCL constraints are defined on a data model. Constraints that are checked are usually related to data used in a sequence flow before this constraint check, and constraints use data from a process instance context.

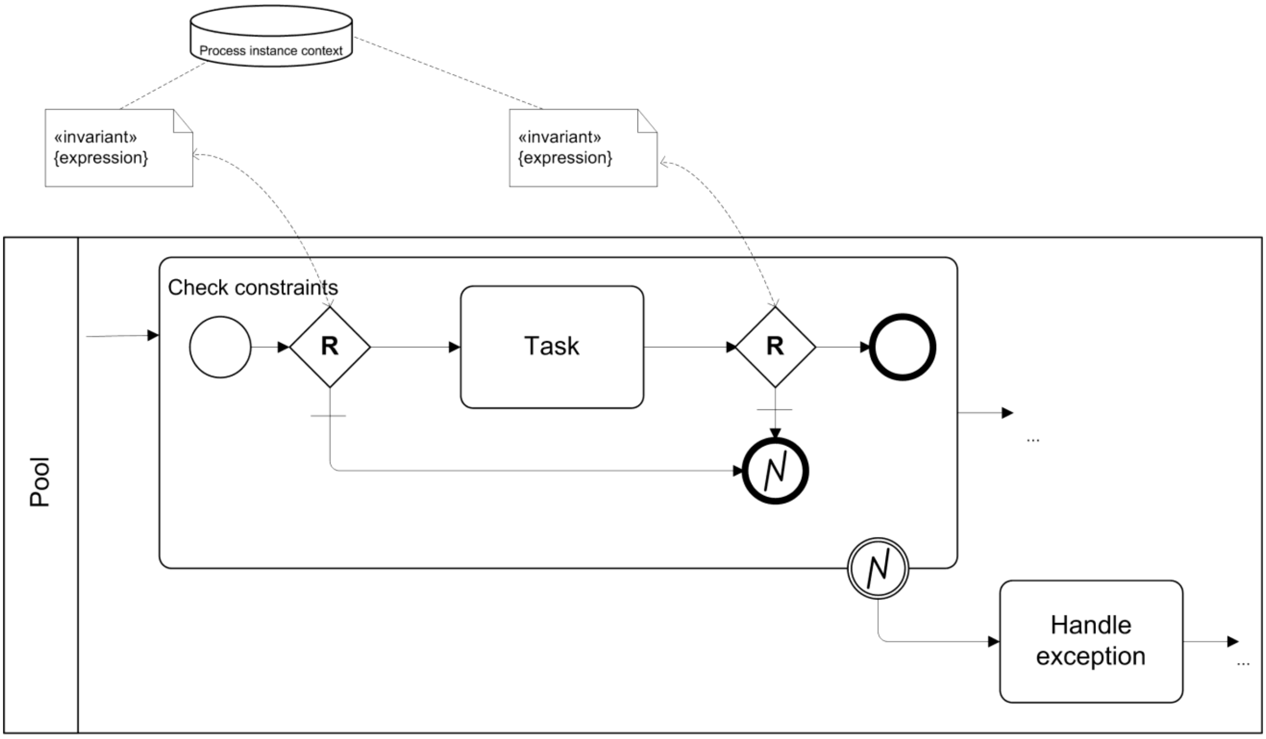


Figure 195. The "Constraints at predefined checkpoint" pattern

4.4.2.2. The "Constraints at multiple checkpoints" pattern

This pattern allows one to dynamically define positions in a business process where constraint checks should be done implying that a single constraint can be checked (i.e., enforced) at multiple places. These places could be before and/or after each activity.

To support this pattern, we define a mapping between a rule gateway, where constraints are checked, and OCL rules that should be evaluated. To retrieve necessary data for the constraint evaluation, constraints need to have an access to business process context (data), that is, there should be a mechanism enabling this constraint enforcement over the business process context. By using an existing rule gateway (similarly to the previous pattern), we avoid introducing new modeling concepts and we also avoid using Aspect Oriented Programming (AOP) as introduced in the original pattern [42], which is not supported by standard business process execution engines. In this pattern, we use the same constraint that is attached to multiple rule gateways in a process, that is, before and after activities. An rBPMN model of this pattern is shown in Figure 196. In the original pattern presented in [43], XSTL is used to add constraint check positions before and end of each activity. Here we used rule gateways in order to connect to the same constraint before and after tasks, so by employing this solution one can define the points in a process where the constraint checks will be done, and without using XSLT and AOP we avoid using additional technologies. In addition, these constraints can be checked by using OWL and SWRL reasoners, as we have mapped OCL and SWRL languages [84].

In Figure 196, rule gateways R_1 and R_2 are attached to the same constraint (invariant), while the rule gateway R_3 is attached to another constraint. In all three cases, the intermediate exception event is invoked in the case when constraint is not satisfied, and in that case the Handle exception task is called to handle the exception. We should note that during the time, some of the rules may change, and due to their declarative nature, not the entire process need to be changed, but only places in a process where those rules apply.

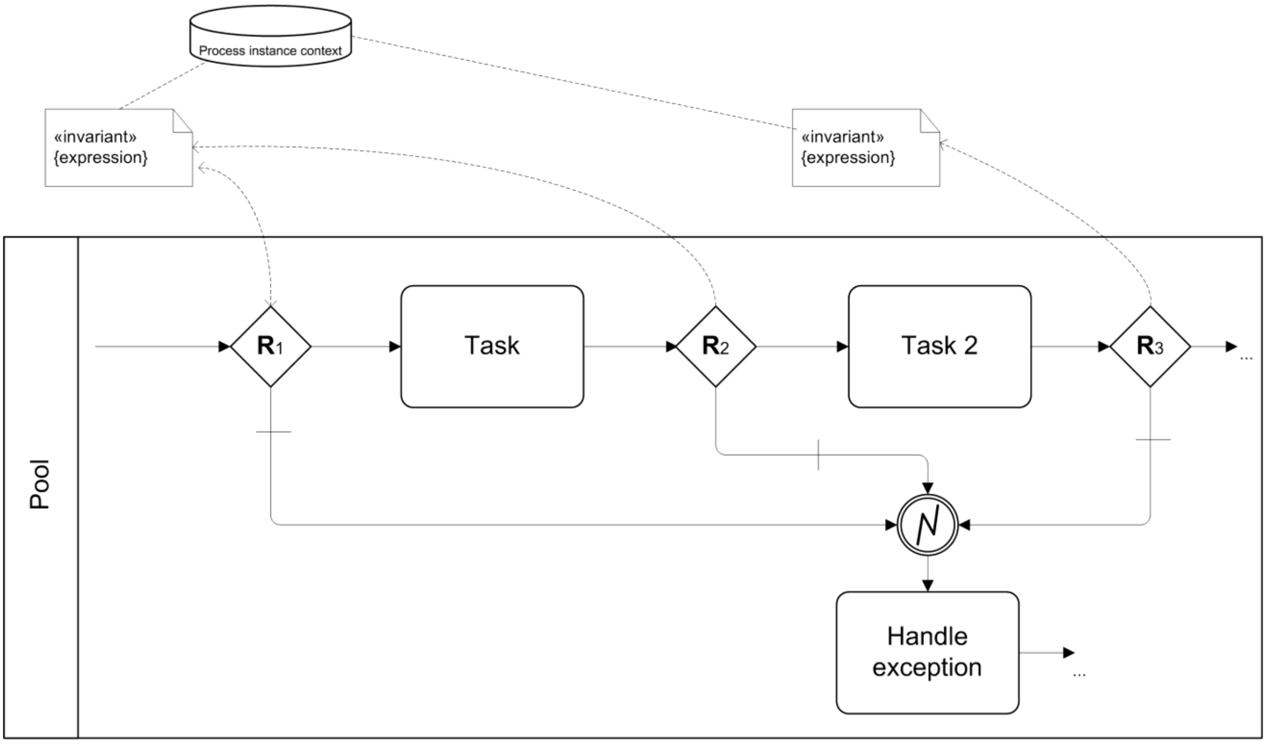


Figure 196. The „Constraints at multiple checkpoints“ pattern

4.4.2.3. The "Constraints enforced by external Data Context" pattern

In this pattern, the business process data context is externalized, so a change in data can be made perceivable by an external applications or a business process that triggers the evaluation of business rules [42]. The business process and business rules can be modified without a need to update each other. The only common points are shared data objects. This implies that the business process must be halted during the business rule evaluation to prevent inconsistencies. This is a challenging task because the business process can execute tasks asynchronously. The other important thing is that any data change triggers the evaluation of all business rules. Therefore, we need a solution to optimize business rules execution by evaluating only relevant business rules.

An important feature of this pattern is to monitor internal business process data. This needs to be supported by business process execution engines. Every change of data would trigger the business process evaluation, and the business process would be halted during the business rule evaluation and would be resumed after a possible constraint violation handling. The main advantage of this approach is to enforce constraints without any modification of the business process.

We model this pattern in rBPMN as shown in Figure 197. We use another business process (External process pool) to act as an external business process context upon which constraints can be validated. In this pattern, we propose replacing the task before which the data should be checked, with the subprocess that contains that task (as a placeholder). The subprocess called "Check constraints on Task" uses the message event to send needed data (or reference to complete process data context – by sending the *processID*) to the External process, which then receives data by using the Receive data task. Note that the message event can be located after the activity, too. When the data is received by the Receive data task, the integrity rule attached to the rule gateway R_1 is used to check whether the data constraint is violated or not. If not, the Store data task is called and the process ends. Otherwise, the Handle violation task is invoked to handle the violation on data.

In order to check constraints in different places in a process, the process designer can manually replace each task with the Check constraints on the Task subprocess, or it can use ATL to automatically replace each task with the subprocess in a business process model.

This pattern have two variants, variant A when the constraint violation is handled by a business process (this variant is shown in Figure 197), and the variant B when the business rule can be used to handle violations. In variant B, we can use another rule gateway with an attached reaction rule, instead of the Handle violation task in the External process pool, where that rule can be used to change business process data.

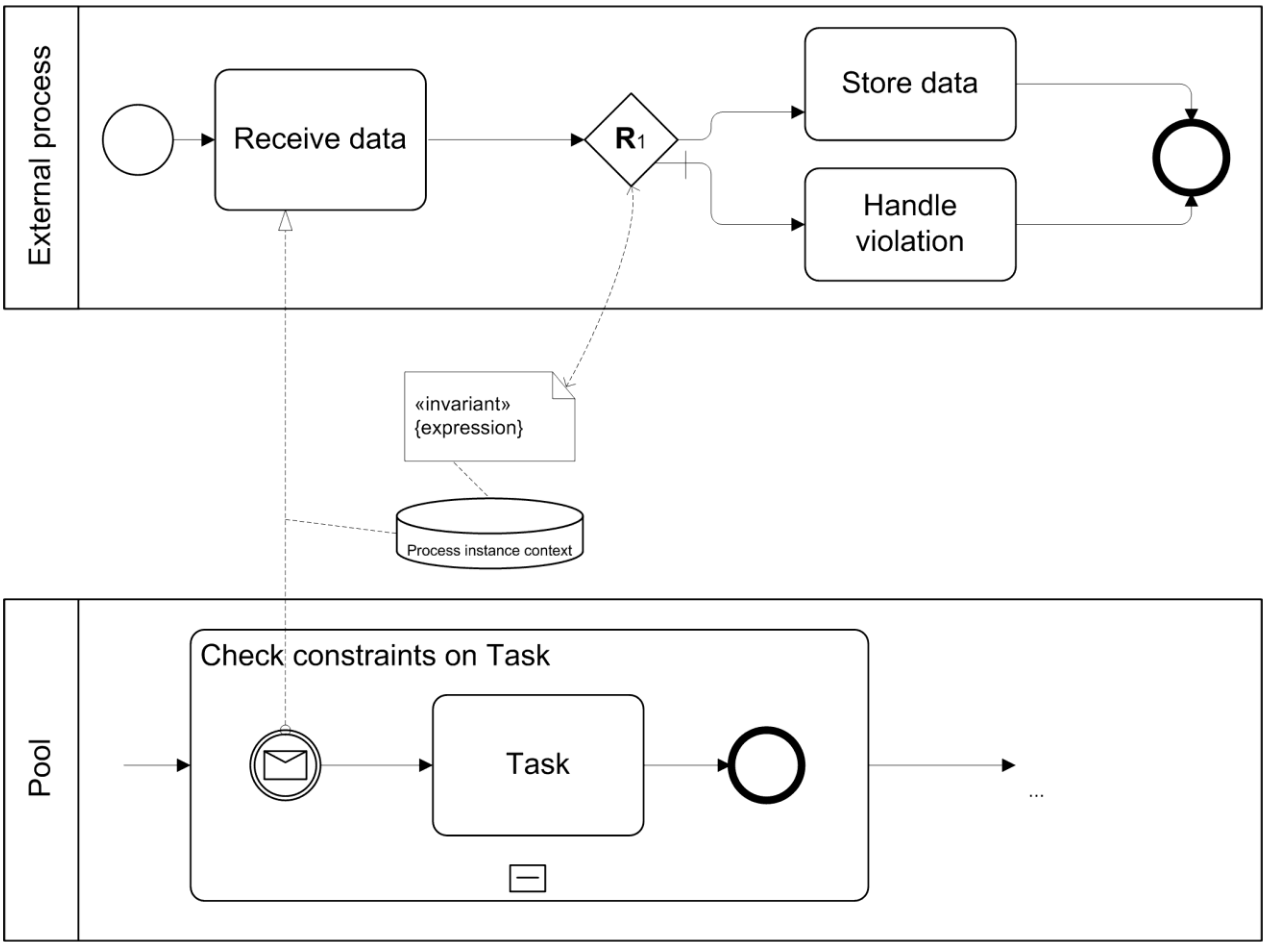


Figure 197. The "Constraints enforced by external Data Context" pattern

An important aspect of this pattern, which contributes to its agility, is that the constraints depend only on data objects, and not on process elements. This means that additional constraints can be added, without the change of main business process during runtime.

4.4.3. Dynamic Business Process Composition

These patterns provide solutions to modeling a business process with the use of process rules to enable dynamic business process composition during runtime. These rules enables to dynamically change of a business process execution, i.e., to select different parts of a business process or to assemble process fragments. The corresponding rules in R2ML are reaction and production rules, as they can be used to invoke tasks or subprocesses.

4.4.3.1. The "Business rule-based subprocess selection" pattern

The purpose of this pattern is to use rules to dynamically select a specified part of a business process depending on the current context of the business process. This pattern is applicable if in the business process, a diverse control flow option exists at a well defined position; and if so, new options might be required in the future. This pattern enables one to externalize these control flow options in subprocesses and to use business rules to make the selection logic agile. This makes it possible to dynamically add new subprocesses during business process runtime.

This scenario is similar to the Decision with flexible output pattern, where for the Business rule-based subprocess selection pattern we select (create) a subprocess instead of an activity. In this pattern, process rules are expressed as production rules attached to the rule gateway. These rules contain needed selection criteria to choose the appropriate subprocess. The condition part of the rules is based on the process context data that is pulled from the process. The action part of the rules returns a subprocess name. The agility provided by this pattern is that the subprocess selection criteria can be changed dynamically during runtime and new subprocesses can be defined after the main process has been deployed.

This pattern is shown in rBPMN in Figure 198. As with the Decision with flexible output pattern, each outgoing sequence flow from a rule gateway is connected to the each production rule. The condition defined on an outgoing sequence flow is mapped to the corresponding rule condition. When rules' condition evaluates to true, a corresponding sequence flow is chosen.

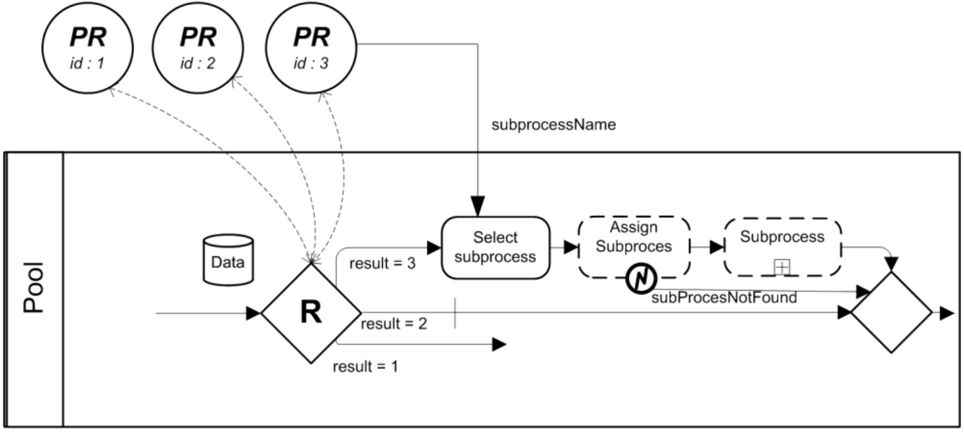


Figure 198. The "Business rule based sub process selection" pattern

4.4.3.2. The "Business Rule based Process Composition" Pattern

This pattern is important especially for long running processes, where unforeseeable circumstances can occur, so a predefined (static) sequence flow is too inflexible. Some parts of the standard business process stay the same, some are omitted under special circumstances and some parts have to be inserted additionally for a business process.

The use of business rules allows for automating the dynamic assembly of the business process parts (called process fragments) [43], but also to interfere with the execution of running business process instances by defining process rules that apply to a single business process instance. The main idea of this pattern is to divide the overall business processes into separate process fragments, where these process fragments can be assembled in different orders. It is also possible to add or define new process fragments during runtime. To be referenced by business rules, the process fragments do have an unambiguous identifier, and they share a common context to work on.

Business rules are used to define which process fragment has to be executed and under which conditions. To enable parallel execution of process fragments, the process rules can be expressed as derivation and production rules attached to a rule gateway. The derivation rules in its conclusion can reutrn a process fragment name, which is then used by a production rule to dynamically invoke that process fragment. The whole process is located in repeating subprocess, as the process fragment invocation is done while predefined conditions are satisfied (e.g., *processRunning* variable is *true*).

We represent process fragments in rBPMN as pools (Process 1 and Process 2), in Figure 199. The rule gateway uses derivation rules which based on predefined conditions in their conclusion part have a next process fragment name to be executed, which is then executed by a production rule (by using the Invoke action). In addition, production rules can invoke multiple fragments in parallel.

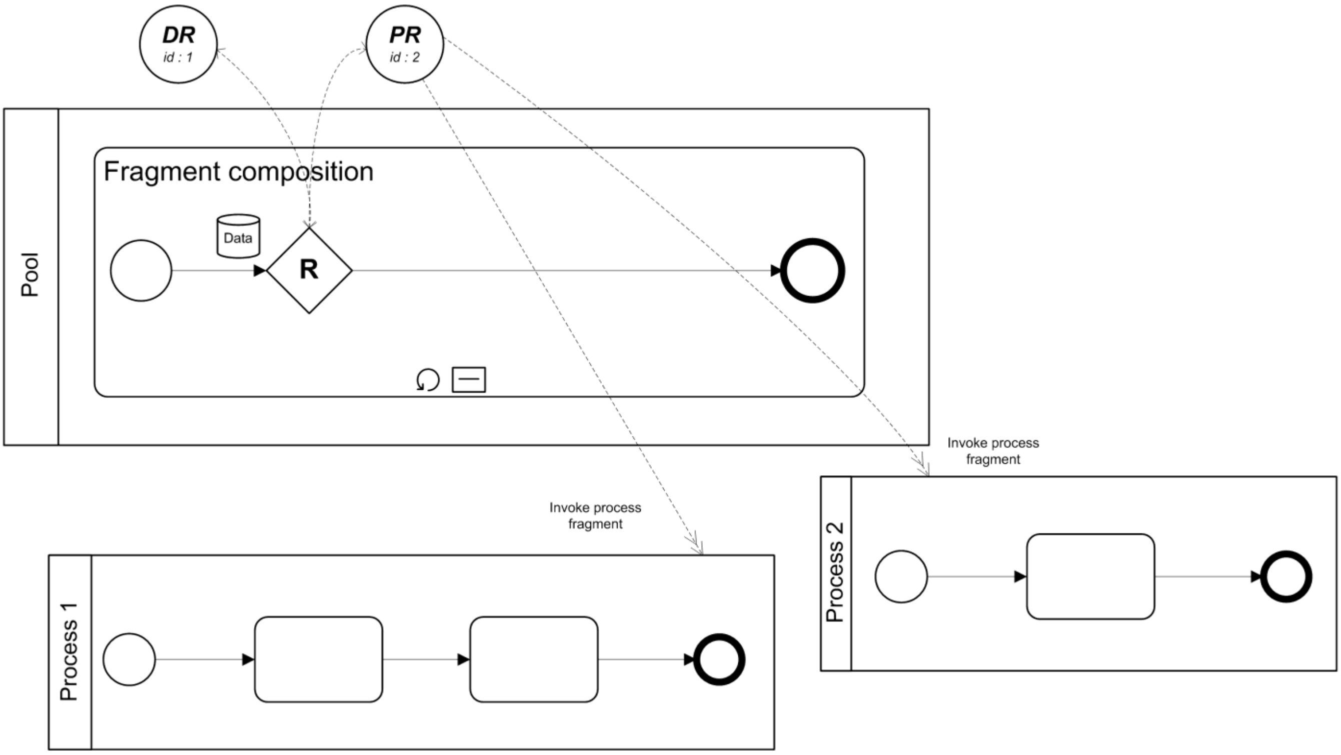


Figure 199. The „Business rule based process composition“ pattern

An important aspect of this pattern is that enables process agility by dynamically changing the process fragment execution order. This is achieved by changing or adding new derivation rules to the rule gateway or to a ruleset. It should also be noted that this pattern has one main disadvantage, and that is the by assembling the process from fragments we may lose an overall picture of the entire business process.

5. Implementation of the rBPMN Language and Case Studies

In this section, we show briefly how the rBPMN language can be used to model service orchestrations, service choreographies and how it can be used in modeling agility business processes in different case studies. All case studies in this section are developed by using our methodology presented in section 3.3, and we will describe them in such methodological way.

5.1. Modeling service orchestrations in rBPMN language

In order to show how rBPMN language can be used in modeling service orchestrations, we will use it in an on-line order scenario, by following the steps from our methodology presented in Chapter 3.3.

Step 1 (*Requirements specification*). In a basic high-level process of the on-line order process, a customer requests a product with some quantity to buy from a seller. The seller then checks whether the requested product is available in stock. If that is a case, the seller creates an order and sends it to the customer for approval and payment. Otherwise, the seller informs the customer that the product is not available. If the customer accepts to pay for the product, the customer proceeds with the payment (by using some payment service) and sends the message to the seller about the payment. Then, the seller chooses a carrier for the paid product, based on some criteria such as the due date, available shipping quantity, and price. When the carrier is selected, the customer is informed about the requested product delivery, and the carrier and the customer proceed with the delivery process. From these requirements, we identified three parties involved: customer, seller and carrier(s), and that we have a certain number of tasks and messages exchanged between these parties, as explained in the steps below.

Step 2 (*Process design*). In this step, we define a high-level process model, based on the requirements. Variability points are defined as abstract activities and later modeled by using rules, depending on level of variability. In Figure 200, we show a business process model in rBPMN. The process is started by a customer (presented as a pool in rBPMN, i.e., Web service), which sends a product order request to the online seller, by using the *Send product order request* task. This request is sent in a form of a message that contains the product name and the requested quantity. When the seller receives the request from the customer by using a message start event, the seller checks whether the requested product quantity is available, and returns that information to the customer. If the product is available, the seller creates the order by using *Create product order* task and sends an order approval message to the customer by using the *Confirm product order* task. When the customer receives these messages, in the case that the seller returned that the requested product is not available, the process ends. However, if the product is available, the seller sends a message to the customer. The message carries information about the available quantity and price, based on which the customer decides whether to buy the product or not. If the received message about the product availability contains the wanted price and the quantity for the customer, the customer decides to buy the product (by using the *Buy product* task). Otherwise, the customer aborts the order (by using the *Abort order* task) and the process ends.

If the seller receives a confirmative message, the seller invokes the *Delivery request* subprocess to find a carrier that will ship the product. Otherwise, the *Seller* suspends the order by using the *Suspend order* subprocess. During the carrier selection, the seller goes through the list of carriers (represented as a multi-instance pool) and checks each seller (by using the *Send availability request* multi-instance task) whether it is available to deliver the quantity of the requested product to the customer in a given period of time. The carrier process is very simple. After reception of the availability request message from the seller, the carrier just uses the *Check availability* task to decide whether it can ship the requested quantity of the product to the customer or. Based on this decision, the carrier returns a corresponding message to the seller. Then, the seller uses its own logic to decide whether to select that carrier. If the carrier is selected, the seller informs the customer about the delivery by using the *Send delivery info* task, and the process continues between the carrier and the customer, where the carrier ships the product to the customer.

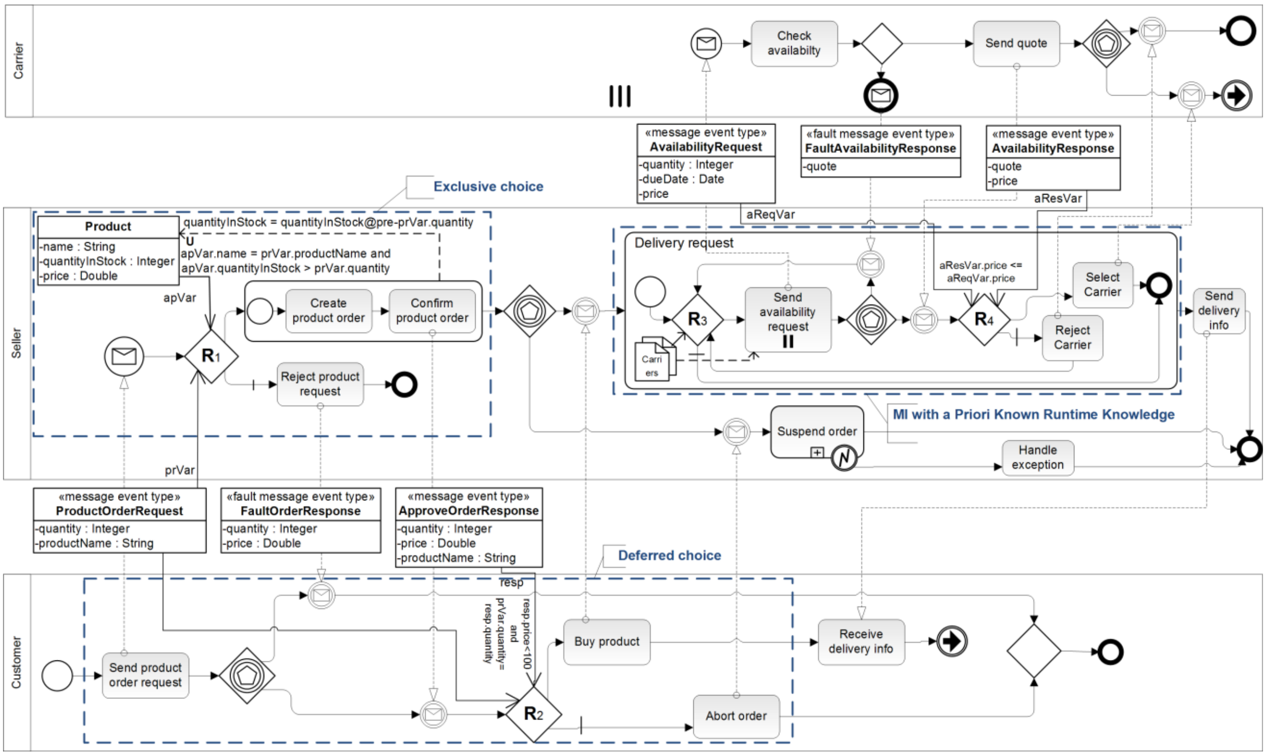


Figure 200. The on-line product order process in rBPMN language

After the high-level definition of the process, we need to identify variability points and to estimate level of their variability. We identified four activities as variability points: i) *Seller decision whether it have a product in the stock*; ii) *Customer response decision, i.e., is he going to buy a product*; iii) *looping through the list of carriers*; and iv) *Seller decision based on a requirement whether carrier satisfies requested conditions* (prior to the carrier selection). We should note that the *Check availability* task in the *Carrier* pool is also a variability point, but for the sake of conciseness, we do not describe this point in detail. The activities that we identified as variable cannot be directly represented in an executable environment, because of their complex nature. When we have identified variability points, we need to estimate the level of their variability. Based on this level, we can decide whether identified variability points will be modeled by means of business rules.

For the first variability point, *Seller decision whether it have a product in the stock*, the outgoing flow from this point depends on the contents of this activity and customer request. In addition, additional logic can be incorporated in this step to give a discount to the customer depending on the requested quantity. The customer can be registered at the seller, and the customer can consequently have an approved access to the further task. This makes the estimated frequency of change high, and as the source of changes is internal, this point can be modeled by using rules. The second variability point, *Customer response decision*, should also be modeled by using rules, as the customer may decide to buy a product based on a offer from the seller (price, discount, warranty, etc.), but it can also try to check the offer for the same product at different sellers. The third variability point, *Looping through the list of carriers*, is not a simple activity. This is the case because the seller should send an availability request to the list of carriers, and based on remaining carriers in the list, which can complete the request, the seller should send the availability request or end the subprocess, if the list is empty. However, the loop through the list of carriers could include only one type of carriers and the list of carriers can be dynamically updated. This means that the frequency of change is high, and the source of change is internal. Thus, this point should be implemented by using rules. The fourth variability point, *Seller* *decision based on a requirement whether carrier satisfies requested conditions*, should be modeled by business rules, because its implication of a change cannot be easily understood. This is the case, as the seller can include more complex carrier selection, based on the number of carriers, combination of product shipment in certain area, available quote, etc., and also to update these conditions during the process execution.

After we have the estimated level of variability, we need to identify workflow patterns for recognized variability points. For the first variability point, we choose the *Exclusive choice* pattern. With this pattern, we can model a selection process of one of several activities based on a control input data (message) and a given condition. A similar case is with the fourth variability point. The selected workflow pattern for the second variability point is the *Deferred choice*, as we need to choose one of several possible branches based on the seller's decision and a given condition. The third variability point is modeled with the *Multiple Instances with a Priori Known Runtime Knowledge* pattern, because we have to loop through multiple carriers and the number of carrier instances is variable. We discuss these patterns in detail in the rest of the section.

Step 3 (*Data design*). In this step, we design underlying data objects (messages), by means of the rBPMN vocabulary. We use these messages to annotate message flows between involved partners. This annotation allows us to use messages in activities and rules as an input and output. We annotate regular messages with "message event type", and fault messages with "fault message event type". In the on-line product order process, we have two logical groups of messages sent, one between the customer and the seller and another one between the seller and the carrier. Regarding the first group of messages, the process starts when the customer sends a *ProductOrderRequest* message to the seller. In this message, we need to have a product name (as String) and a product quantity (as Integer). This message is represented with the *prVar* variable used in the process model. As a response from the seller, we have two output message flows, one when order is created and send to the customer for approval, and another one when the product is not available. If the product is not available, the seller sends the *FaultOrderResponse* message to the customer, with the available quantity and the price, so that the customer can decide whether to buy the available quantity. If the product is available, the seller creates an order, and the *ApproveOrderResponse* message is sent to the customer. This message is represented with the *resp* variable in the process. This message contains all important information about requested product: quantity, price and product name. Based on this information, the customer can decide whether to buy the product or not. On the other hand, the seller sends the *AvailabilityRequest* message to a carrier in order to check if that particular carrier is available to ship the requested product. This message contains the quantity, dueDate and desirable shipment price for the selected product. The *AvailabilityRequest* message is represented with the *aReqVar* variable in the process The carrier returns two messages, the *FaultAvailabilityResponse* in the case when it cannot ship the product, and the *AvailabilityResponse*, when it can ship the product in due date (including the quote and price of a product shipment). The latter message is represented with the *aResVar* variable in the process model.

Step 4 (*Rule design*). When the process diagram elements and messages are defined, we need to implement rules for each identified variability point. We model the first variability point, *Seller decision whether it have product in stock*, by using reaction rules attached to the rule gateway (R_1). We use reaction rules, because we have a message event as its trigger and we want to define a triggered action. These rules, based on the defined condition evaluates whether the requested product is available at the seller's side or not. We have here two reaction rules (see Figure 201): one with a positive condition, and another one with the negated condition. This is the case due to the nature of reasoning of rules, where the use of ELSE statements in rules may lead to reasoning problems [110][142]. If the product is not available at the seller's side in the requested quantity, the reaction rule with the negated condition is activated and the *Reject product request* task is executed, and followed by the *FaultOrderResponse* message sent to the customer. However, if the condition of the reaction rules attached to the rule gateway (R_1) evaluates to true, the subprocess that contains the *Create product order* and the *Confirm product order* tasks is selected (we denote the negative condition output flows with a crossed line in

URML diagrams such as Figure 201). The resulting activity from this flow is the *ApproveOrderResponse* message, which is sent to the customer by using the *Confirm product order* task. Figure 201 also shows how we achieved traceability between rBPMN and R2ML elements. That is, all BPMN tasks (*Create product order*, *Confirm product order* and *Reject product request*) and messages (*ApproveOrderResponse* and *FaultOrderResponse*) have their corresponding parts in R2ML. The traceability is established through the rBPMN metamodel and the OCL constraints such as those given in Section 4.3.

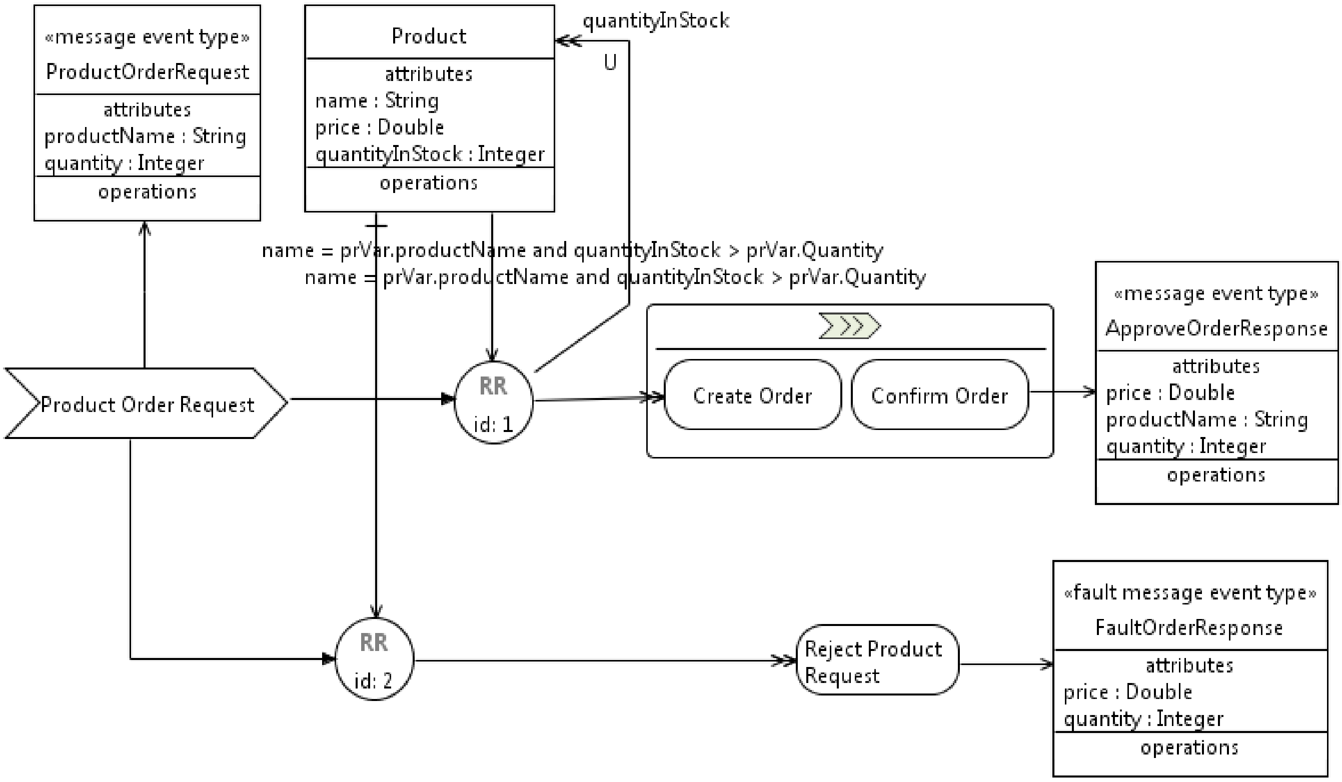


Figure 201. Reaction rules attached to the rule gateway (R_1) in Figure 200

Once the customer has received the *ApproveOrderResponse* message, by using the intermediate message event, the sequence flow goes to the rule gateway (R_2), i.e., the second variability point (*Customer response decision*). The customer uses the rule gateway (R_2) to decide whether to buy the product. This rule gateway also has two attached reaction rules, one with a positive and one with a negated condition (*resp.price < 100 and prVar.quantity = resp.quantity*). These two rules are shown in Figure 202.

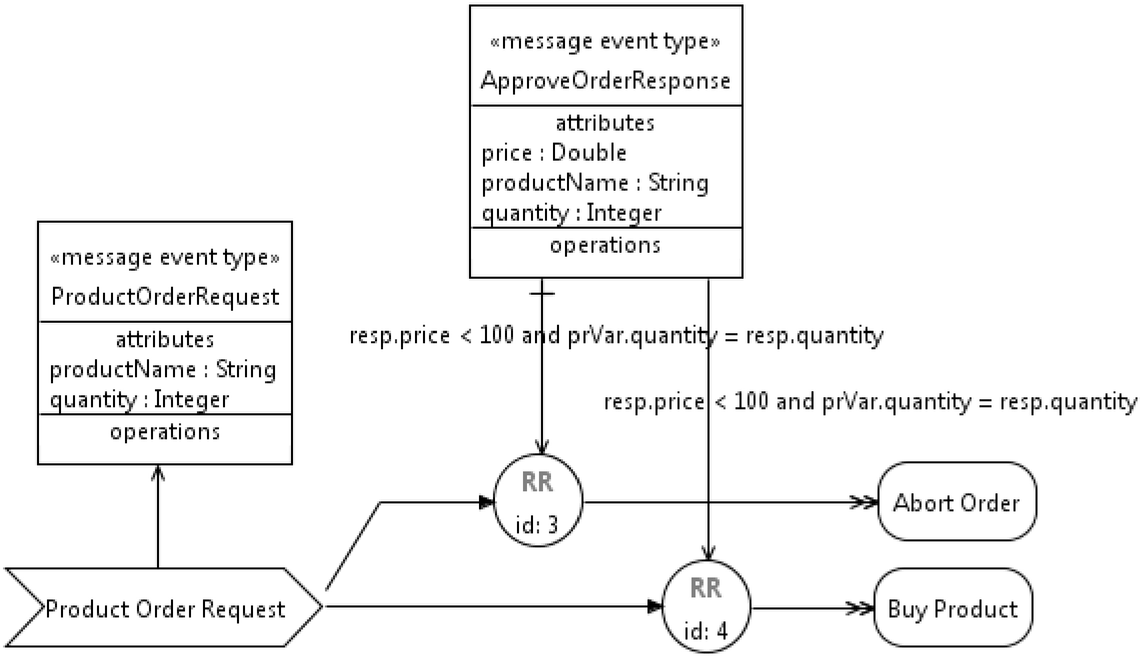


Figure 202. Reaction rules attached to the rule gateway (R_2) in Figure 200

This condition means that the rule gateway's (R_2) outgoing sequence flow will be chosen if the product price is less than 100 units and if the offered quantity is the same as the requested quantity. If the condition is satisfied, the *Buy product* task is used to send a message to the seller to inform the seller that the customer wants to buy the product. The actual *ProductOrderRequest* message is sent, but we omit it from Figure 200 for the sake of clarity. If the condition defined on the rule gateway (R_2) evaluates to false, the *Abort order* task is used to send the message to the seller and the process ends for the customer. Once the seller has receive this message, it starts the *Suspend order* subprocess in order to cancel the order. During the execution of this subprocess, an exception can occur, just like we have shown for the Cancel case pattern (see section 4.3.6.2). In this case, the sequence flow goes to the *Handle exception* task, used to handle this task. On the other hand, once the seller has received the information that the customer wants to buy the product, the seller starts the *Delivery request* subprocess. The subprocess starts with the start event, which follows to the rule gateway (R_3). This is actually our third variability point, *Looping through the list of carriers*. Here, we also employ two reaction rules attached to the rule gateway, one with a positive and another one with a negated condition (we omitted condition from diagram for the sake of simplicity – see Figure 203).

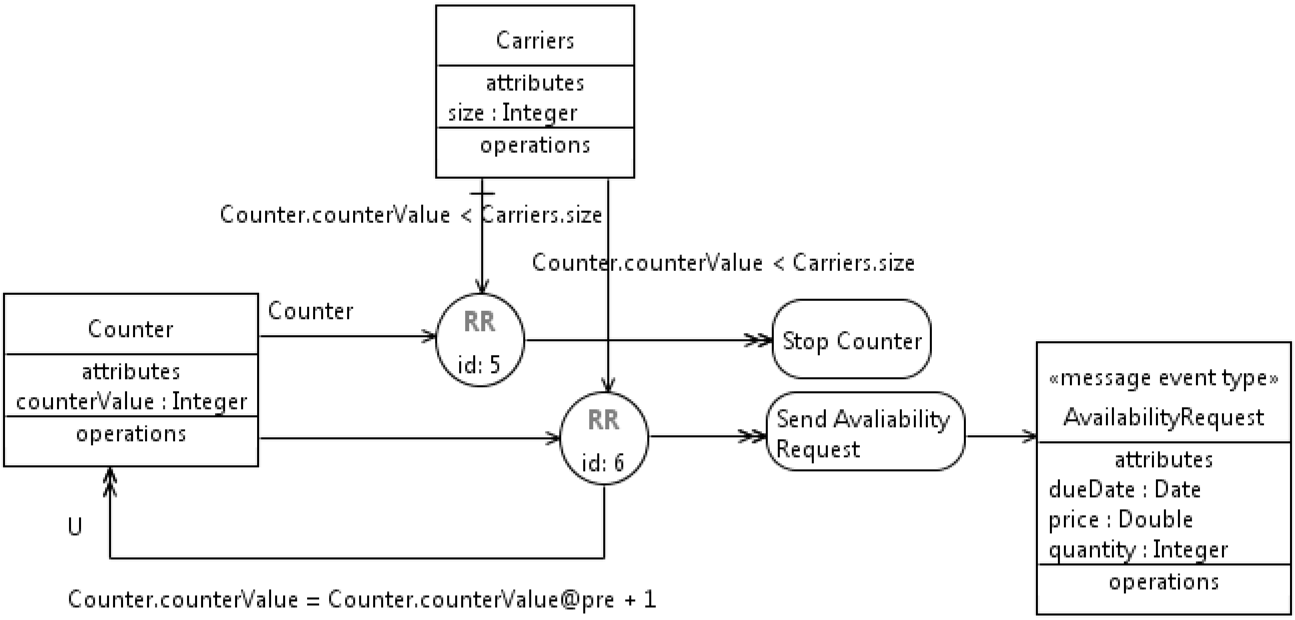


Figure 203. Reaction rules attached to the rule gateway (R_3) in Figure 200

The condition is defined as *Counter.counterValue < Carriers.size*, where after each positive evaluation, we increment the counter by one (by using *Counter.counterValue = Counter.counterValue@pre + 1* expression). If this condition evaluates to true, the sequence flow goes to the *Send availability request* multi-instance task, used to send the *AvailabilityRequest* message to each *Carrier* from the list of carriers. However, if this condition evaluates to false, the subprocess ends. When a carrier receives this message, it uses the *Check availability* task to evaluate whether the requested quantity of the product can be shipped in due time. If it cannot ship the product, the carrier returns the *FaultAvailibilityResponse* message to the seller. Otherwise, the carrier returns the *AvailabilityResonse* message to the seller, with the offered shipment price for the quantity.

Then, we arrive to the fourth variability point, *Seller decision based on requirement whether Carrier satisfies requested conditions*, where we also use two reaction rules attached to the rule gateway (R_4) to check if the shipment price returned from the carrier is less or equal to the requested price or not (see Figure 204). If so, the seller selects the carrier by using the *Select carrier* task and informs the carrier about that decision by sending a message. After this selection the process ends. Otherwise, the carrier is rejected by using the *Reject carrier* task, which is also used to send the appropriate rejection message to the carrier. Then, the sequence flow returns to the start of the subprocess (rule gateway R_3). When the *Delivery request* process ends, the seller uses the *Send delivery info* task to inform the customer about shipment details. Once the customer has received this message, the process of shipment continues between the selected carrier and the customer (we omit this part for the sake of clarity).

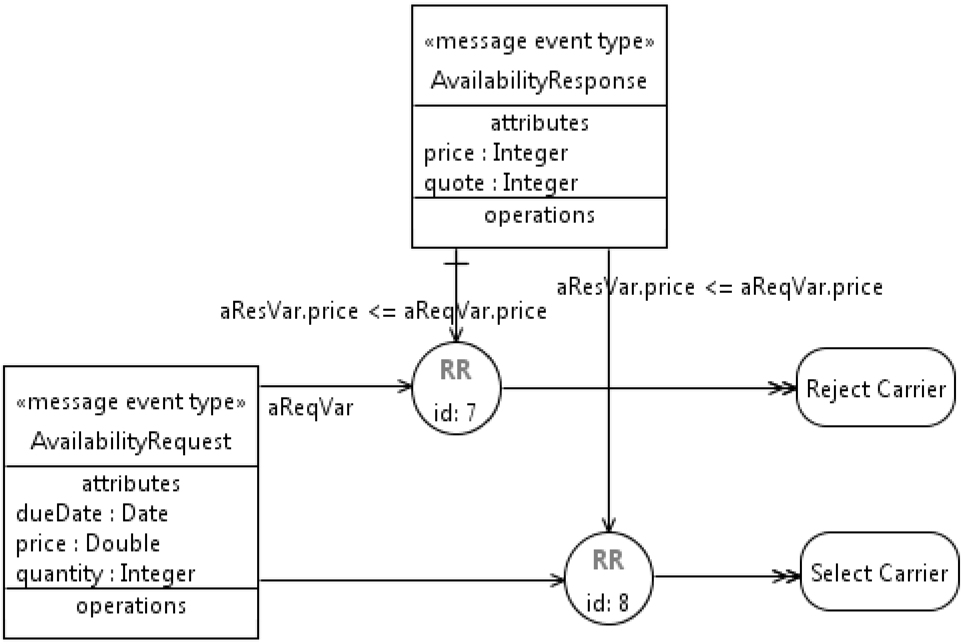


Figure 204. Reaction rules attached to the rule gateway (R_4) in Figure 200

In the process shown in Figure 200, we have identified three different workflow patterns. The first pattern that we identified is the *Exclusive choice* pattern, which we used to define the first variability point in the process. We used this pattern in the *Seller* pool in order to select one of two available activities based on a condition. This pattern defined in rBPMN has two main advantages compared to the standard BPMN solution. First, the rule gateway (R_1) and the subprocess that follows its positive conclusion share the same common vocabulary, i.e., they can access the same *Product*. Second, the content of the *ProductOrderRequest* message, used in rules condition, is not fixed, but can be changed at runtime on every *Customer* request.

In this pattern, we can also recognize a relation between the described process and concrete services that can be used to implement the modeled process, and thus consequent service orchestration. Here, we actually can indentify several message exchanges between services called, and those message exchanges can directly be mapped to the standard Message Exchange Patterns. Here, we have an In-Out MEP. That pattern consists of two messages: a message received by the service from some other node, followed by a message sent to the other node. The In-Out MEP can return a fault message, which in this case is the *FaultOrderResponse* message. This relates to our earlier work where we have defined mappings between reaction rules and Web service descriptions [111], where a reaction rules are used to model a MEP. Triggering rule events are modeled as input messages, while triggered rule events are output messages. As event expressions in R2ML have an event type assigned, we can generate the complete message types (i.e., complexTypes). In addition, we can generate from our reaction rule model implementation in a concrete rule-based language. We have provided a full definition of several languages (e.g., Jess and Drools) by simulating semantics of reaction rules on production rule engines.

We have also used the *Deferred choice* pattern in the *Customer* pool (the second variability point). After sending the *ProductOrderRequest* message, the sequence flow goes to the event-based gateway, with two possible outgoing alternatives (branches). These branches can be selected based on the received message event. When the message is received from the seller, we use reaction rules attached to the rule gateway (R_2) in order to additionally constraint the sequence flow and to introduce new logic to that flow.

When the customer sends a positive response to the seller, the *Delivery request* subprocess begins. In this subprocess, we have the *Multiple Instances with a Priori Known Runtime Knowledge* pattern (the third variability point). The *Send availability request* task is performed for each carrier in the set of carriers. This set can be dynamically changed during the process execution by some reaction or production rule. The execution of this task is enabled by using the reaction rules attached to the rule gateway (R_3).

5.2. Modeling choreographies in rPBMN language

In this section, we give an example of a service choreography modeled by the rBPMN language. The example is about of a flight booking process where a traveler uses an agent to book a flight with a *Travel agency* via the *Trip request* task.

Step 1 (*Requirements specification*). In our flight request process, a traveler agent requests a trip organization from a travel agency. When such a request is received, the travel agency starts with the airliner selection process. In this process, a request for a flight availability and price is sent to each airliner that the travel agency works with. Once received the request from the travel agency, an airliner calculates the price and checks if the requested number of seats is available on the requested departure date. Based on the outcome of this check, the airliner sends a message to the travel agency that a flight is not available or sends the flight is available (with price for a seat). Using this infromation that is collected from airliners, the travel agency selects an airliner with the lowest price and contacts the airliner for reservation. After this, an e-ticket is issued to the traveler. If no airliner has available seats for the requested departure date, the traveler is informed about that. From the requirements, we identified three parties involved in the process: travel agent, travel agency and airliner.

Step 2 (*Process design*). The process starts when the *Travel agency* receives a request from the Traveler agent. In Figure 205, we show a business process model in rBPMN. Once the request is received by the *Travel agency*, the “Request price” subprocess is started. In this subprocess, the *FlightRequest* message is sent to each *Airliner* from the *Airlines* participant set, by using the “*Request flight price*” task. The message request is a message, which specifies the requested *departureDate*, *arrivingAirport* and *seats* attributes. When this request is received by the Airliner through the start message event, the *Calculate price* task is used to calculate the price for the requested *departureDate*, *arrivingAirport* and number of *seats*. If the number of seats left for the *Flight* on the requested departure date is less than the number of seats requested, then the the *Send airline not found* task is invoked; this is followed by the message sent *FaultFlightResponse* to the *Travel agency*. This message contains a fault description. If the number of seats left for the Flight is equal to the numbers of seats requested, the sequence flow which goes to the *Send Flight Price* task is selected. The *Send Flight Price* task is used to send the message that containts the flight number and price to the *Travel agency*. When all of the *Airlines* are contacted, the *Request price* subprocess ends. The the *Travel agency* checks whether we have any *Airliners* that satisfied conditions. If so, the *Select Airline* task is invoked; otherwise the *Send airline not found* task is invoked to inform the *Traveler agent* that no flights can be found for the requested departure date, arriving airport and requested number of seats. In the *Select airline* task, the *Airliner* is selected and the message is sent to the *Airliner* to reserve the seats on the flight. When the message is received by the *Airliner*, by using the intermediate message event, it updates the number of the *seatsLeft* on the *Flight*. Then, the *Make reservation* task is invoked, which creates the reservation and sends it to the *Travel agency*. When the *Travel agency* receives the message, it prepares and sends an e-ticket to the *Traveler agent*.

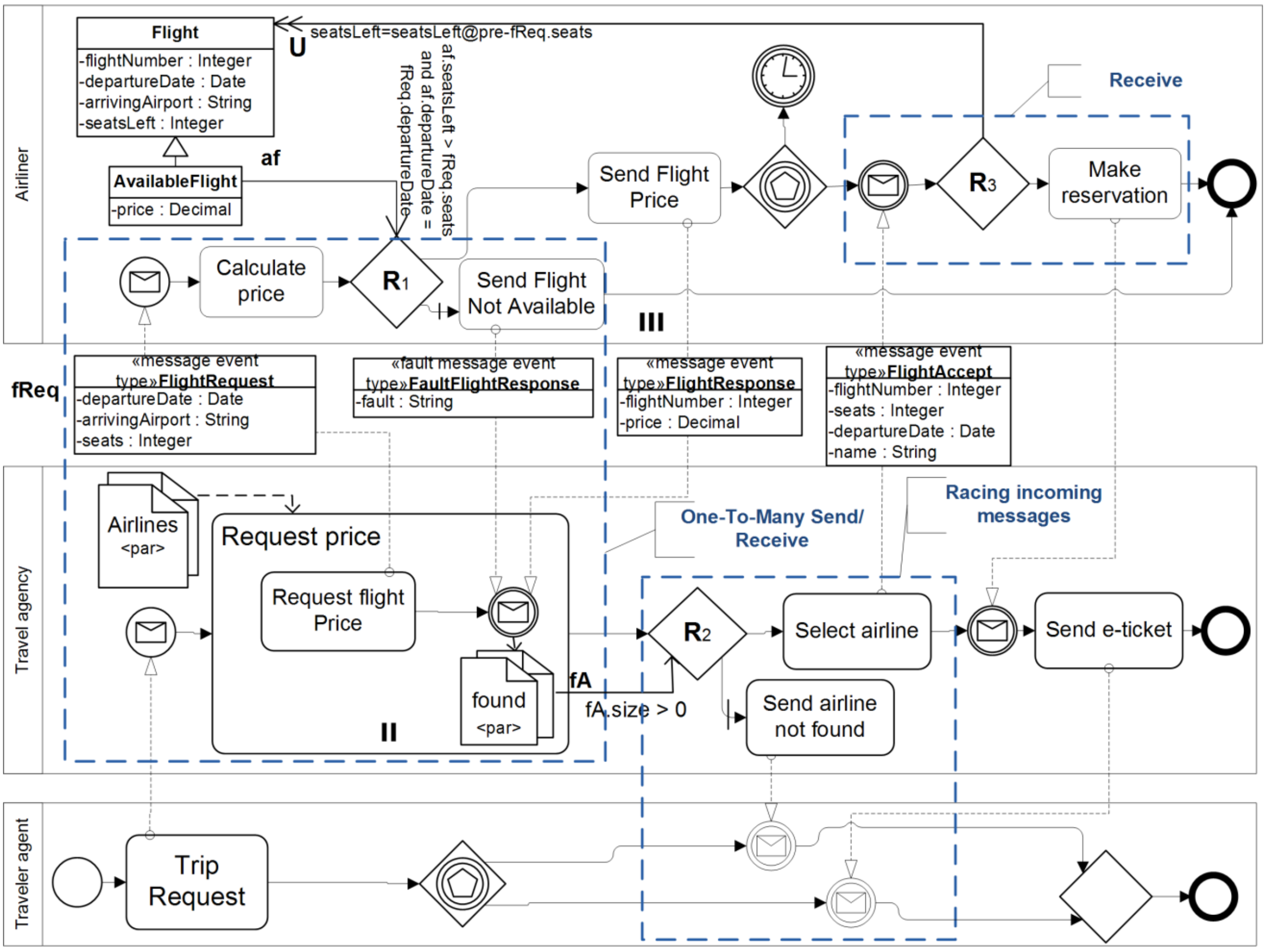


Figure 205. Interconnected behavioral choreography diagram for the Flight request process in rBPMN language

When we define the overall process, we need to identify variability points in the process. In the Flight request process, we identified three variability points: i) *Airliner decision whether flight is available*, ii) *Selection of airliner*; iii) *Airliner update of a current flight state*. When we identified the variability points, the next step is to estimate their level of variability. Based on their variability, we can infere which ones should be implemented by business rules.

Regarding the first variability point, *Airliner decision whether flight is available*, the process flow forks at this point according to the available flight and additional requests. For this point, the source of change is internal, and the frequency of changes is high, because the airliner can give additional discounts based on the number of seats requested. The process flow also forks according to the departure date (e.g., if the departure date is away the discount is given). Based on this, this point should be modeled by business rules. The second variability point, *Selection of airliner*, should be modeled by rules, since the source of change is internal. In this particular case, at this variability point, we check whether the airliner that fullfills the request is found. However, at this place, the *Travel agency* can include additional conditions, for example, to filter airliners or to send additional requests to the airliner in order to collect more information. The third variability point, *Airliner update of a current flight state*, should also be modeled by business rules, because the source of change is internal, where in this point the Airliner could decide not only to update the seats left, but also to accept additional requests from a Travel agency. Another implication of variability here is that we can check if the user is a regular user and if so, to give him a discount.

The next step is to identify appropriate service interaction patterns for these variability points. For the first variability point, we choose the *One-To-Many Send/Receive* pattern. In this pattern, a participant sends out several requests to other different participants and waits for their responses. Here, we have the multiple-instance *Request price* subprocess with the *Request flight price* task that is used to send messages to the *Airliners*. For the second variability point, we can recognize one variant of the Racing incoming messages pattern. The *Trip request* task is followed by an Event-based gateway, from which two possible branches could be selected based on the received message. The third variability point is modeled by the *Receive* pattern. When the message is received from the *Traveler agency*, and handled by using the intermediate event message, we need to decrease the number of seats left on the flight. In this point, after message reception, the Airliner can decide to follow multiple paths depending on the request.

Step 3 (*Data design*). In this step, we need to define underlying data objects (i.e., messages) used in a proces. As noted in Section 5.1, we annotate message flows with messages, so that activities and gateways can have input and output data. Those messages are regular (marked with "message event type" stereotype) or fault (marked with "fault message event type" stereotype). In the concrete case of the Flight request process, the first message is sent as a flight price request from the travel agency to the airliner. This message is called *FlightRequest* and contains *departureDtate*, *arrivingAirport* and *seats* attributes, which are used in the process model. We have two messages as the response from the airliner. The first message is sent when a flight is available. This message is called *FlightResponse* message, and it contains *flightNumber* and *price*. If the flight is not available the *FaultFlightResponse* message is sent to the travel agency, and this message contains just the *fault* description. When the airliner is selected, the *FlightAccept* message is sent to the airliner, in order to confirm the reservation. This message has the flight number, number of seats, departure date and passenger name attributes.

Step 4 (*Rule design*). In this step, we need to define concrete rules and corresponding process elements for each identified variability point. The first variability point, called *Airliner decision whether flight is available*, we modeled by using a rule gateway (R_1) and two reaction rules attached to it (see Figure 206). We use reaction rules, because we have a message event as a triggering event and an activity as a triggered event. These rules are based on pre-defined conditions, which evaluate whether there are any free seats for the flight at requested *departureDate*. One rule has a positive condition and another has the negated condition. If the number of *seats* left for the *Flight* on the requested departure date is less than the number of seats requested, then the reaction rule with the negated condition is activated and the *Send airline not found* task is invoked; this is followed by the message sent *FaultFlightResponse* to the *Travel agency*. This message contains a fault description. If the condition on the reaction rule attached to the rule gateway (R1) evaluates to true, the sequence flow which goes to the *Send Flight Price* task is selected. In this case, the message flow is annotated by the *FlightResponse* message. As in the case of workflow patterns (see Section 5.1), we achieved here the same traceability between rBPMN and R2ML elements by using rBPMN metamodel.

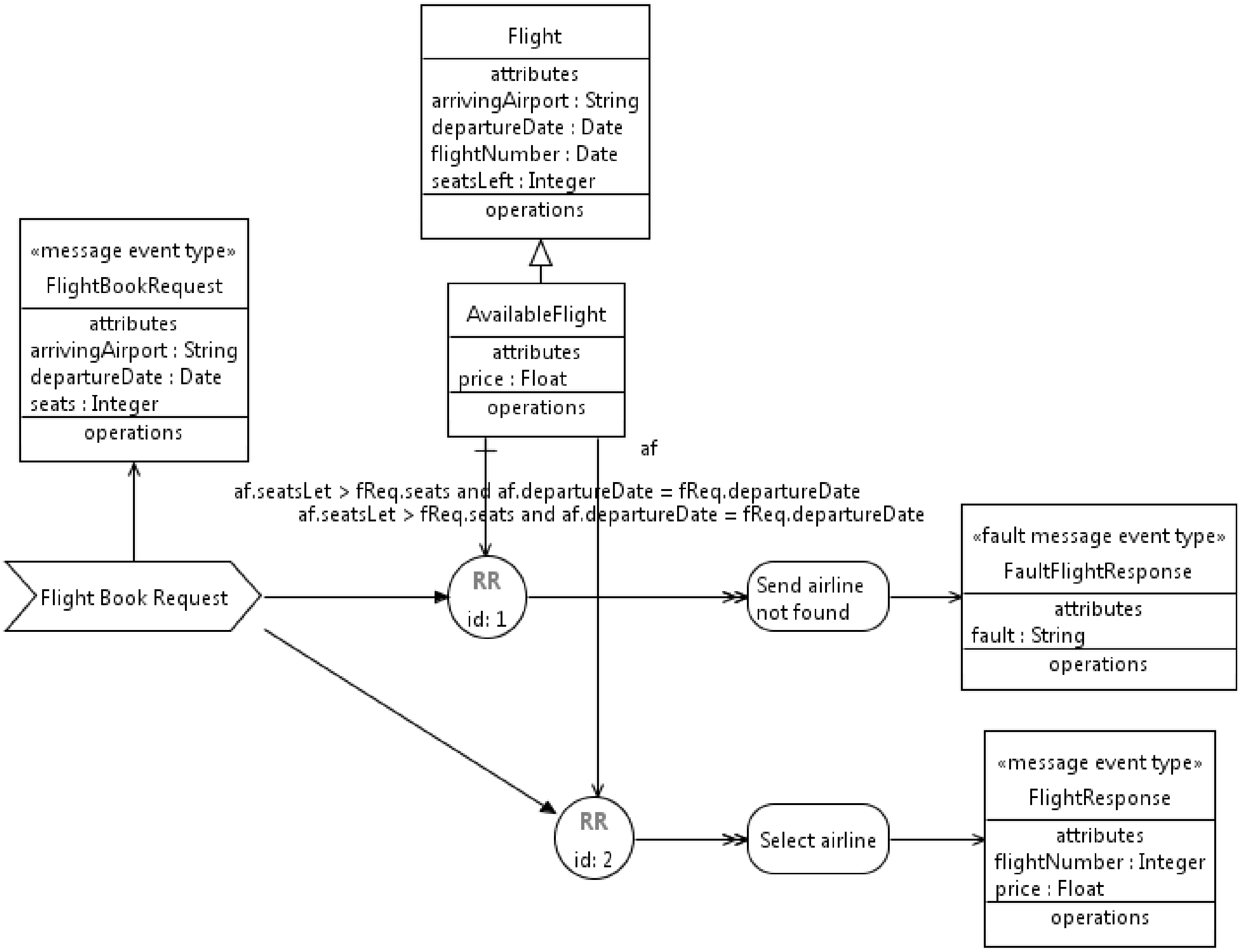


Figure 206. Reaction rules attached to the rule gateway (R_1) in Figure 205

When the *FlightResponse* message is received by the *Travel agency*, it writes a reference to the *Airliner* which sends the message to the *found* participant set. When all of the *Airlines* from the *Airlines* participant set are contacted, the *Request price* subprocess ends and the sequence flow goes to the R_2 rule gateway. This rule gateway actually represents our second variability point (*Selection of airliner*). We use this rule gateway to check whether we have any *Airliners* in the *found* participant set. If so, the *Select Airline* task is invoked; otherwise the *Send airline not found* task is invoked to inform the *Traveler agent* that no flights can be found for the requested departure date, arriving airport and requested number of seats. We also have two reaction rules attached to the R_2 rule gateway (see Figure 207). In the *Select airline* task, the *Airliner* is selected and the message is sent to the *Airliner* to reserve the seats on the flight.

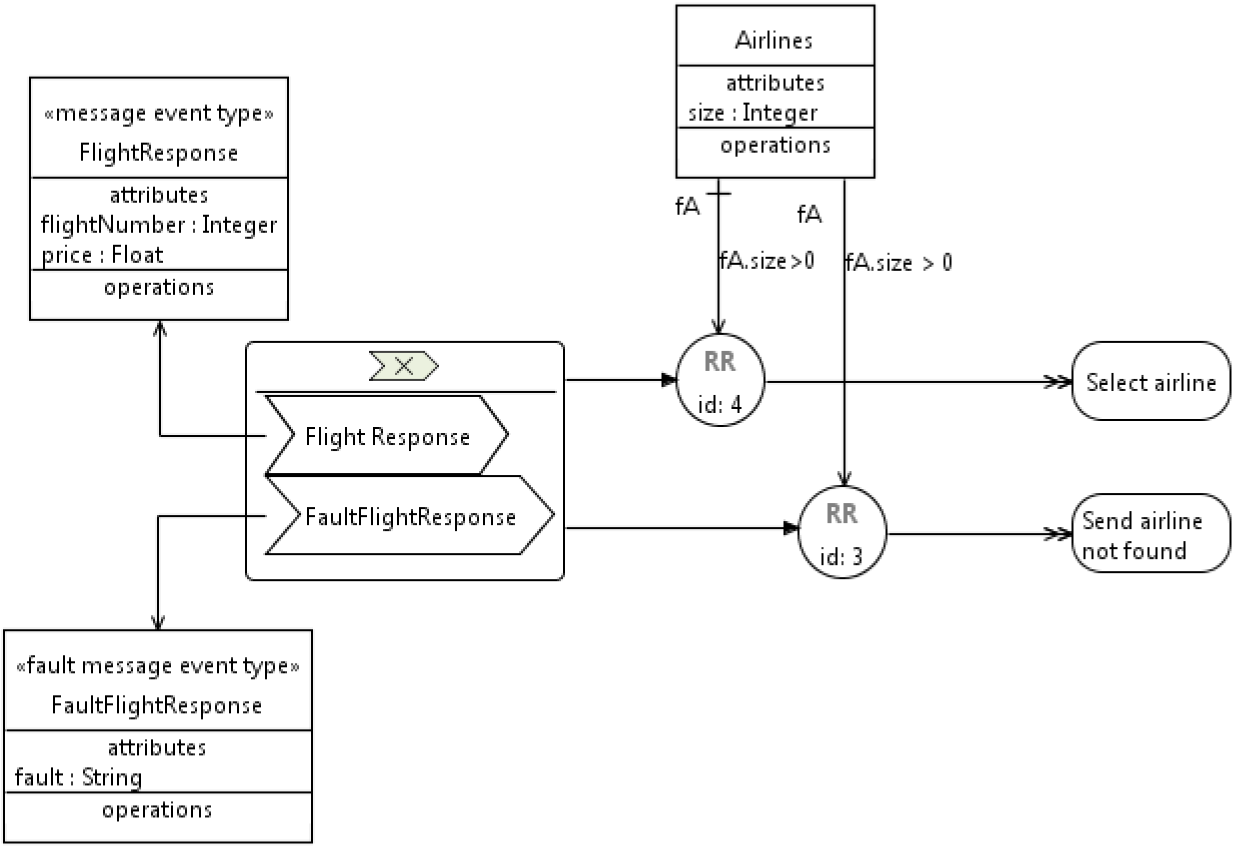


Figure 207. Reaction rules attached to the rule gateway (R_2) in Figure 205 represented in URML

When the message is received by the *Airliner*, by using an intermediate message event, the reaction rule attached to the rule gateway R_3 is invoked to update the number of the *seatsLeft* on the *Flight* by using the update action (U) on the attached reaction rule (see Figure 208). This represents our third variability point, *Airliner update of a current flight state*.

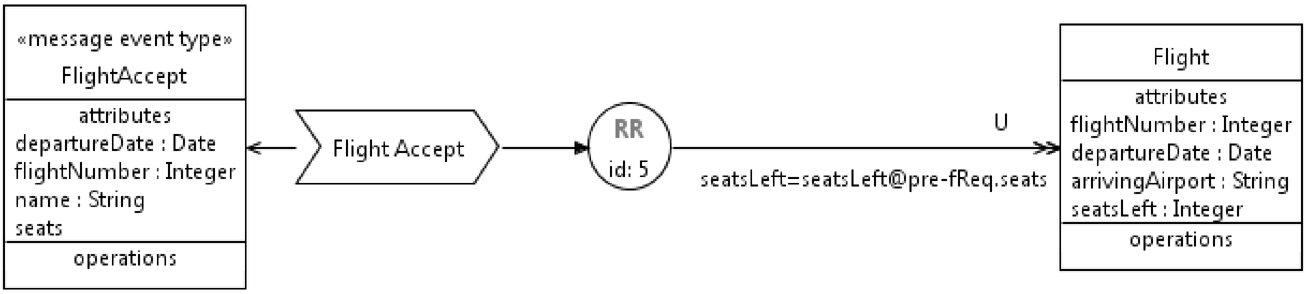


Figure 208. Reaction rule attached to the rule gateway (R_3) in Figure 205

In the process presented in this section, we identified three service interaction patterns. The first identified pattern is the *One-To-Many Send/Receive* pattern. Here, we have the multiple-instance *Request price* subprocess with the *Request flight price* task that is used to send messages to the *Airliners*, by using the information about partners from the participant set (*Airlines*). The reason for such a design decision is because the number of partners may or may not be known at design time and this represents an advantage of our approach. Another advantage of our approach is the shared vocabulary based on which two rule gateways R_1 and R_3 can access the same *Flight* instance to check or decrease the value of their *seatsLeft* attribute. Also, *seats* and *departureDate* values used in the rules' condition are not fixed, and can be dynamically changed in each request of the *Travel Agency*, by using the information from the *FlightRequest* message type and *Flight* class, respectively.

One another important aspect of this pattern is related to the In-Out Web service Message Exchange Pattern (MEP), as we can transform reaction rules into complete Web service descriptions [111]. That allows for transforming a (set of) R2ML reaction rule(s) model(s) into a MEP. Triggering events model input messages, while triggered events are output messages. As all event expressions in R2ML have an event (and, thus message) type assigned, we can generate the complete message types (i.e., complexTypes) from our reaction rules. Another important implication of our model is that for each reaction rule in R2ML, we can also generate its implementation in a concrete rule-based language.

In the *Airliner* pool, we can see the *Receive* pattern. An important implication of using rules here is that we can attach another rule to the rule gateway to check if the user is a regular user and if so, to give him a discount. Given the nature of rules, if the discounting business logic changes, this can be dynamically reflected by the change of the rule. Finally, in the *Traveler agent* pool, we can recognize one variant of the Racing incoming messages pattern. Here, we can use rules to introduce additional logic, such as to check if the *Traveler* has enough money on the account to pay the ticket.

Following our methodology, using interconnected behavioral models is not suitable for modeling choreographies, so we need to translate such models into interaction models. We first defined an interconnected model in order to better understand a coreography from the perspective of each participant. Then we can translate such model into an interaction model, as choreographies are not based on individual views on choreography, but on the global perspective. In Figure 209, we show an interaction model of the choreography for the flight request process. Modeling choreographies in this way has two major drawbacks, as reported in [22]: redundancy (where parallelism, branching, loops and timeouts are duplicated in the model) and potentially incompatible behavior (errors in the model in the case of event-based XOR-gateways). Interaction models do not have these problems, so we translated our model to the interaction model and shown in Figure 209. In the interaction model, we attached a message event to the each message interaction, while the pools are empty. In the interaction models, we need to connect the rule gateway to the participant (pool), so that the participant can invoke the rule and decide which branch to take. In the case of Event-based gateways, one of multiple events can happen, by occurring first in the process. Another implication of the interaction models is passing the participant references by using the participant sets. In the process model shown in Figure 209, we need to collect the *Airliner* participants in the *found* participant set, in order to pass that participant set to the rule gateway (R_2) condition. A participant set (*found*) is attached to the message flow and each time when the flight price message is received, the participant is written in the set.

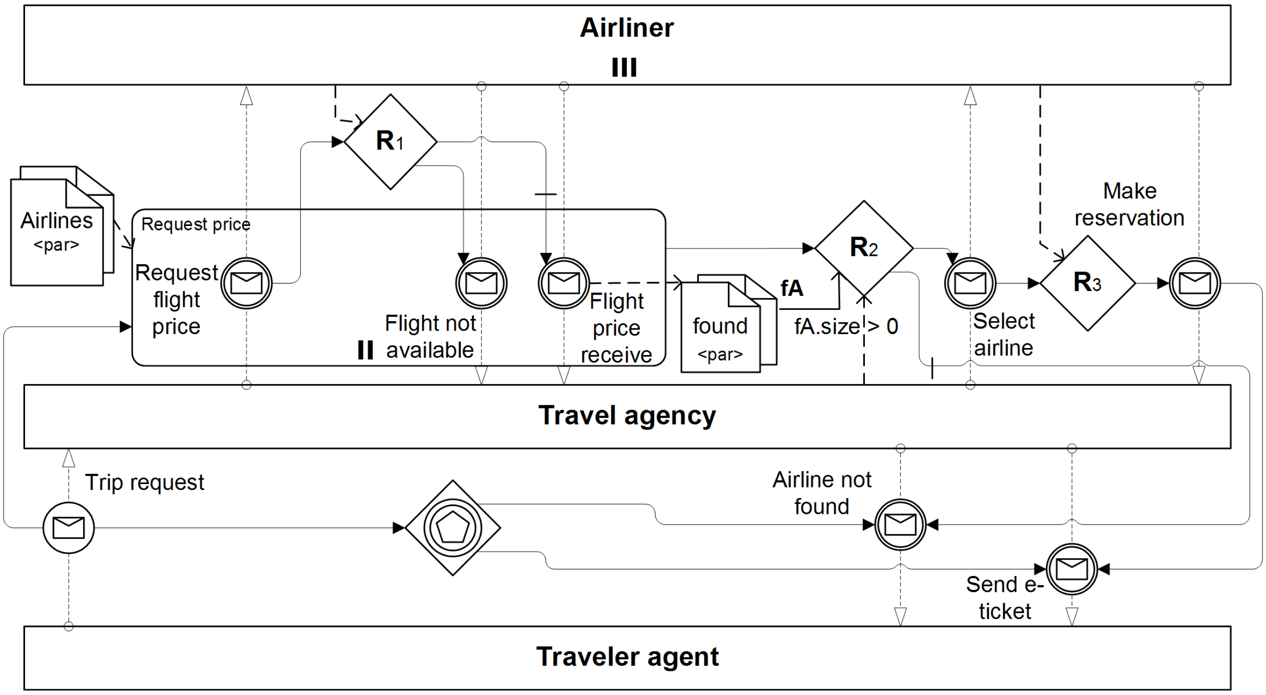


Figure 209. rBPMN interaction choreography model for the flight request process show in Figure 205

From this section we can see the following contributions for modeling choreographies in rBPMN:

- definition of message types exchanged between parties involved in the process, i.e., connection with structural models;
- definition of message exchanges between rules, which enables:
  - dynamic changes of a business process, i.e., dynamic flow change because of rule declarative nature;
  - complete generation of Web service descriptions (with conditions);
  - definition of conditions on which some interaction can occur, as well as constraints;
  - modeling of complex events.
- compliance of business processes with respect to the business policies and constraints:
  - compliance between orchestrations and choreographies, where rules defined in orchestrations can be used in choreographies, too;
  - generation of choreographies from orchestration models, i.e., traceability between elements in orchestrations and choreographies.

5.3. Modeling Agile Business Processes in the rBPMN Language

In this section, we show how the rBPMN language can be used for modeling more agile business processes (orchestrations), in terms defined for agile patterns (see section 4.4). In order to show how agile patterns can be used in modeling of business processes, we employ a book buying use case. We will present this use case using our methodology from section 3.3.

Step 1 (*Requirements specification*). In this process, a customer requests a book to buy from a bookstore. When such a request is received by the bookstore, it checks whether the book is available and sends a message to inform the customer. In this point, a discount to the customer is calculated in the offered price. The customer then decides whether he is going to buy the book or not. If the customer decides not to buy the book, the process ends; otherwise, the customer sends a message to the bookstore to buy the book. Then, the bookstore decrease the number of the books in stock, calculates a discount to the customer (if any in applicable), checks if the customer is a silver or gold customer, and uses this information to send the book and receipt to the customer. When the customer receives the book, he has seven days to decide whether he wants to hold the book or not due to various reasons such as a wrong book has been received. If he decides to return the book to the bookstore, he sends a mail to the bookstore. In the mail, he can request either his money to be returned back or a correct book to be sent. In the former case, the payment is rollbacked and the money is returned to the customer, and in the later case, a new copy of the book is sent, where the customer again has seven days to decide if he wants to hold the book or not. From this requirements specification, we can conclude that we have two parties involved: the customer and the bookstore.

Step 2 (*Process design*). In this step, we design the overall process model, based on the requirements specification. In Figure 210 we show a business process model in the rBPMN language. The process begins when the *Customer* logs in and requests the information about the book that (s)he wants to buy from a *Bookstore*. This request is modeled by using the *Book request* task that sends a message from the *Customer* to the *Bookstore* (represented by the *Bookstore* pool). This message request contains the requested quantity of the books and the book name. When such a request is received by the *Bookstore*, the Bookstore evaluates whether the requested book in the requested quantity is available in store. If the book is not available in the *Bookstore* or there is no requested quantity, the *Send book not available* task is preformed and is followed by a message sent to the *Customer*. This message contains the information why book is not available. When the customer receives such a message the process ends. If the book is available the process flow goes on to the *Calculate discount* task, which is used to calculate the discount depending on the *Customer* status (e.g., gold or silver user) and then to the *Send book available* task, which is used to send the message to the customer that the requested book is available. This message contains the price of the requested book. When this message is received by the customer, the *Customer* uses an activity to decide whether (s)he will buy the book with the offered price. If the *Customer* decides to buy the book, the *Buy book* task is used to send a message to the *Bookstore* to inform that the customer wants to buy the book. This message is the same as the first message sent in the process. The *Bookstore*, by sending the message after the *Send book available* task, uses an Event-based gateway to wait for the customer's decision for 24h (by using intermediate message timer event). If the message confirming that the *Customer* wants to buy the book is not received within 24h, a timeout event is generated and the process ends. However, if the *Customer* decided to buy the book, the *Bookstore* uses an activity to decrease a quantity in stock for the requested book and to issue a receipt for the *Customer* based on his status. After that, the *Send book* task is performed to send the book to the *Customer*, who receives the message by using the *Receive* book task, and the process ends. Then, the *Customer* has 7 days to decide whether he wants to return the book or not (e.g., if he received the wrong book). If the *Customer* decides to return the book, he sends an email to the *Bookstore*. When such an email is received by the *Bookstore*, depending on the *Customer* request, a new copy of the book is sent to the *Customer* or the payment is rollbacked and the money is returned to the *Customer*.

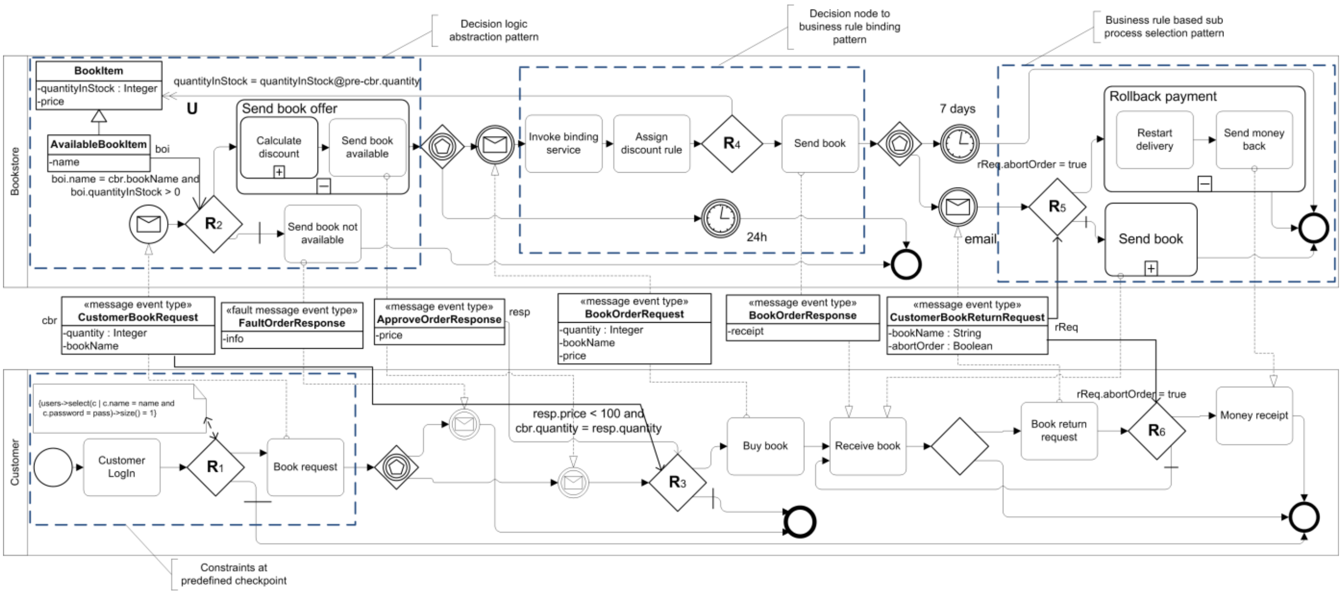


Figure 210. The book buy request scenario in rBPMN language

After defining a high-level definition of the process, the next step in our methodology is to identify variability points in the process. In this process we identified following variability points: i) *Customer login*; ii) *Bookstore decision whether the book is available*; iii) *Customer response decision, i.e., is he going to buy a book*; iv) *Bookstore update of a book stock quantity*; v) *Customer decision to return a book*; vi) *Bookstore rollback payment decision*; and vii) *Customer decision to abort the order*. When we have estimated the variability points in the process, we need to identify the level of their variability and later we can decide whether those variability points need to be modeled by means of business rules.

For the first variability point, *Customer login*, the user must login to the system in order to buy a book. Thus, this activity has the internal source of change and it should be implemented by using business rules. In addition, this activity has an organization-wide scope, i.e., the login activity usually needs to be done in multiple processes of the same organization. The second variability point, *Bookstore decision whether the book is available*, has an outgoing sequence flow that depends on the contents of this activity and on a customer request. In addition, in this step, the Bookstore could decide to give a discount to the customer depending on the requested quantity or the customer's status. As the source of change is internal and the frequency of change is high, this point should be modeled by means of business rules. The third variability point, *Customer response decision, i.e., is he going to buy a book*, should also be modeled by using business rules, as the customer may decide to buy the book based on the *Bookstore* offer (e.g., discount or price). So, in this case, the source of change is internal and frequency of changes is high. The fourth variability point, *Bookstore update of a book stock quantity*, requires an update of the book quanity in stock, as well as issuing a receipt based on the discount given to the customer. As the source of change in this case is internal, this variability point should be modeled by business rules. The fifth variability point, *Customer decision to return a book*, implies that the customer can decide to return the book based on her/his personal decision (i.e., human decision). Hence, this variability point should not be automated and implemented by using business rules. The sixth variability point, *Bookstore rollback payment decision*, is not a simple activity, the bookstore needs to decide whether to return the book to the customer, or to rollback the payment. This means that the frequency of change is high and the source of change is internal, and that this point should be modeled by business rules. The seventh variability point, *Customer decision to abort the order*, should also be modeled by business rules, a its implication of change cannot easily be understood. This is the case because the customer could involve more complex decision to abort the order, such as to check if the book name is correct, ISBN, or some other relevant information.

Once we have identified the variability points and estimated the level of their variability, we need to identify appropriate patterns for implementing these points. For the first variability point, *Customer login*, we choose the *Constraints at predefined checkpoint* agility pattern (see Section 4.4), because this pattern is used in the process to check predefined constraints, such as to validate data before the activity. The second variability point, *Bookstore decision whether the book is available*, is modeled as the *Decision logic abstraction* agility pattern. By using this pattern, we can define the business logic by using a business rule in the business process. The third variability point, *Customer response decision*, is also modeled by the *Decision logic abstraction* agility pattern, as the process flow is similar to the previous variability point. The fourth variability point *Bookstore update of a book stock quantity*, is modeled by using the *Decision node to business rule binding* agility pattern. This is the case, as we need to use a mapping to bind the decision node to the derivation rule dynamically, because in this case we want to calculate the discount based on the *Customer* status. The sixth variability point, *Bookstore rollback payment decision*, is modeled by the *Business rule-based subprocess selection* agility pattern. By using this pattern, we can dynamically choce the specified part of a business process, that is, in this particular case, that part is subprocesses. In addition, we can dynamically add a new subprocess during execution. The seventh variability point, *Customer decision to abort the order*, should also be modeled by the *Decision logic abstraction* agility pattern, because we need to incorporate some additional logic in a business process.

Step 3 (*Data design*). In this step, we design messages that are exchanged between parties/activities, by using the rBPMN vocabulary. The process starts when the customer sends a *CustomerBookRequest* message to the *Bookstore*. This message containes the book name and qumtity attributes, and it is represented by the *cbr* variable in the business process model. As a response from the bookstore, we have two outgoing message flows, one when the book is available and another one when the book is not available. If the book is available, the *ApproveOrderResponse* message is sent to the customer. This message is represented with the *resp* variable in the process model. This message contains a price of the requested book. Based on this message, the customer will decide whether to buy the book or not. However, if the book is not available, the *FaultOrderResponse* message is sent to the customer. This message contains the price for the book and the available quantity (that can be less than requested quantity). If the customer decides to buy the book, the *CustomerBookRequest* message is again sent to the bookstore. The last message sent in the process is the *CustomerBookReturnRequest* message, which is sent from the customer to the bookstore in the case when the customer decides to return the book. This message contains the book name along with a signal whether to abort the order or not. The message is represented with *rReq* variable in the process model.

Step 4 (*Rule design*). Now, for each identified variability point and corresponding pattern, we need to define rules. We model the first variability point, *Customer login*, by using integrity rules in form of an OCL invariant, attached to the rule gateway (R_1). The invariant is shown in Figure 210, and we defined it as *{users->select(c | c.name = name and c.password = pass)->size() = 1}*, which is used to go through the list of users and to check if there exists a user with a required *name* and *password*. If so, the user is logged in and the process flow goes to the *Book request* task. Otherwise, the process ends. The main advantage of our solution compared to the standard BPMN solution is that we can use a shared vocabulary in the whole process which is in this case for example important for the user login data. Additionaly, we can dynamically add different types of login during the process execution by changing the rule or by adding more integrity rules.

Then, the process flow goes on to the second variability point, *Bookstore decision whether the book is available*, where we used reaction rules attached to the rule gateway (R_2), as we have an event-based request annotated by a message before the rule gateway . These rules based on their conditions evaluate whether the book is available or not. Actually, we have two reaction rules (see Figure 211): one with a positive condtion, and another one with the negated condition (the condition checks if the book with the requested name and quantity exists in the store). If the book is available at the bookstore, the reaction rule with the positive condition is triggered, and the *Send book offer* subprocess with the

Calculate discount and *Send book available* tasks is performed. After that, the *ApproveOrderResponse* message is sent to the customer in the message flow. If the book is not available, the reaction rule with the negated condition is invoked, and the *Send book not available* task is performed. This is followed by the *FaultOrderResponse* message sent to the customer. Each of the reaction rules is attached to the corresponding outgoing sequence flows from the rule gateway (R_2), i.e., rule with negated condition is attached to the crossed outgoing sequence flow.

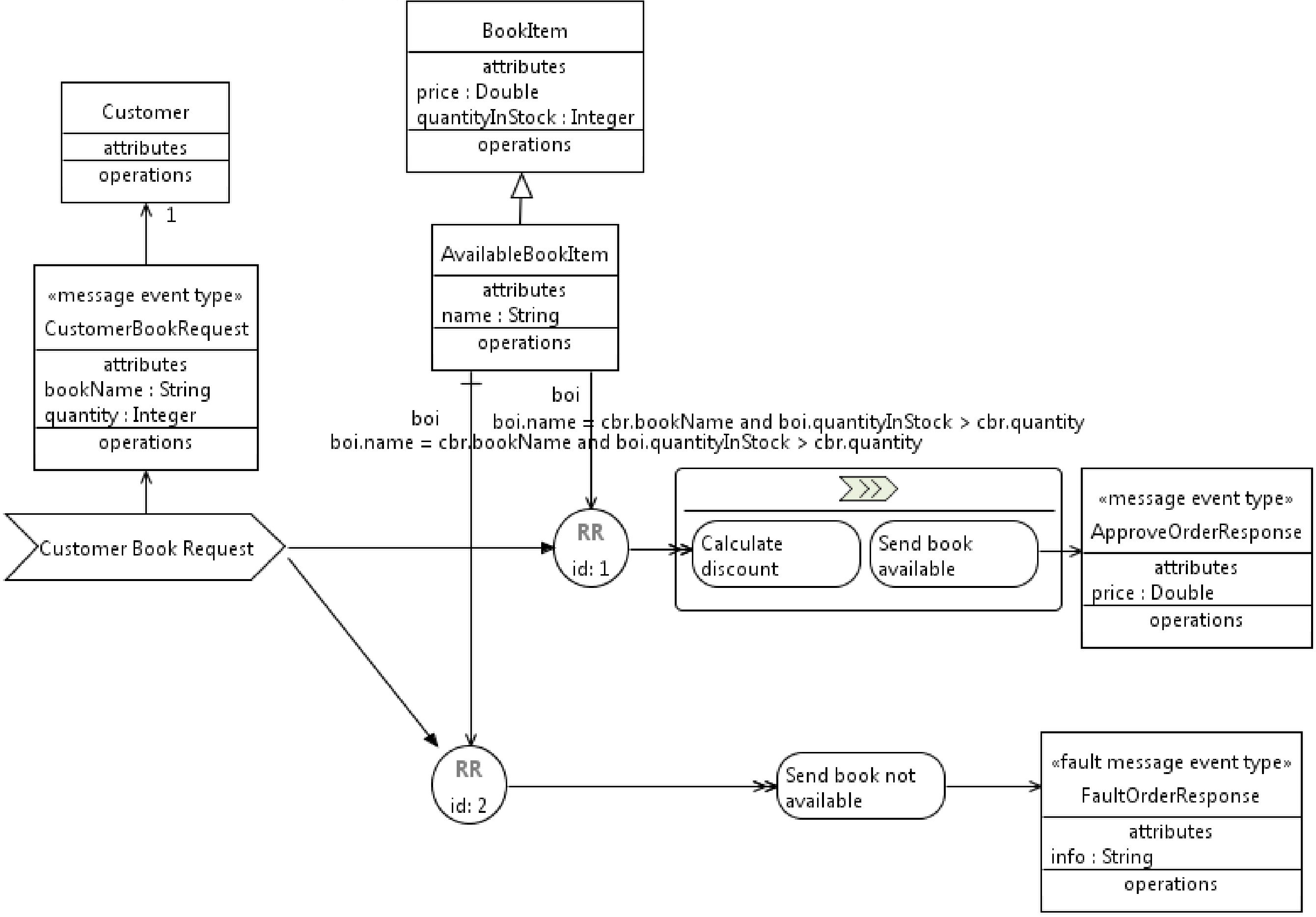


Figure 211. Reaction rules attached to the rule gateway (R_2) in Figure 210

When the customer receieves the *ApproveOrderResponse* message, by using the intermediate message flow, the sequence flow goes on to the rule gateway (R_3), i.e., the third variability point (*Customer response decision*). The customer uses the rule gateway (R_3) to decide whether to buy the book, as we have an annotated message before the rule gateway. This rule gateway has attached two reaction rules on it (see Figure 212), one with a positive condition and one with the negated condition (*resp.price < 100 and cbr.quantity = resp.quantity*). The rule with positive condition will be enabled if the book price is less then 100 and the requested quantity is equal to the quantity offered by the bookstore; otherwise, the rule with the negated condition will be enabled. If the rule is satisfied, the *Buy book* task is used to send a message (*BookOrderRequest*) to the bookstore in order to inform the bookstore that the customer wants to buy the book. If the condition evaluates to false, the process ends.

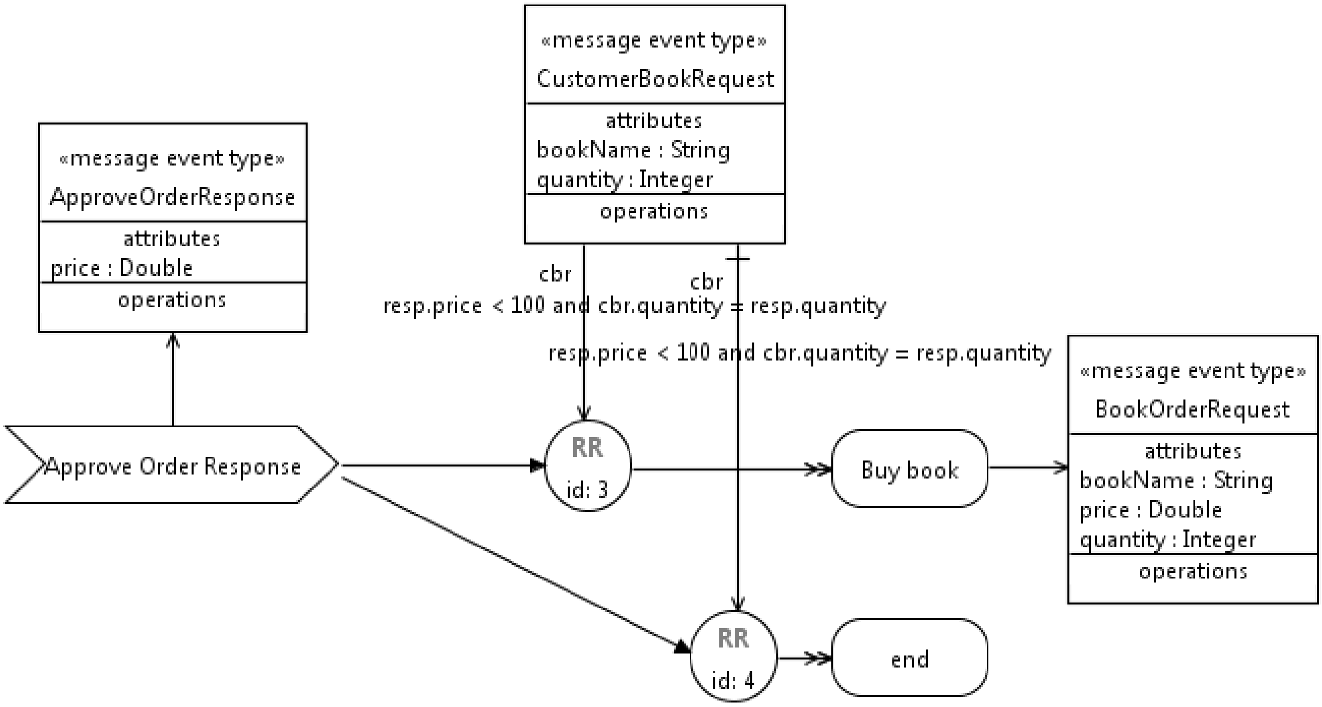
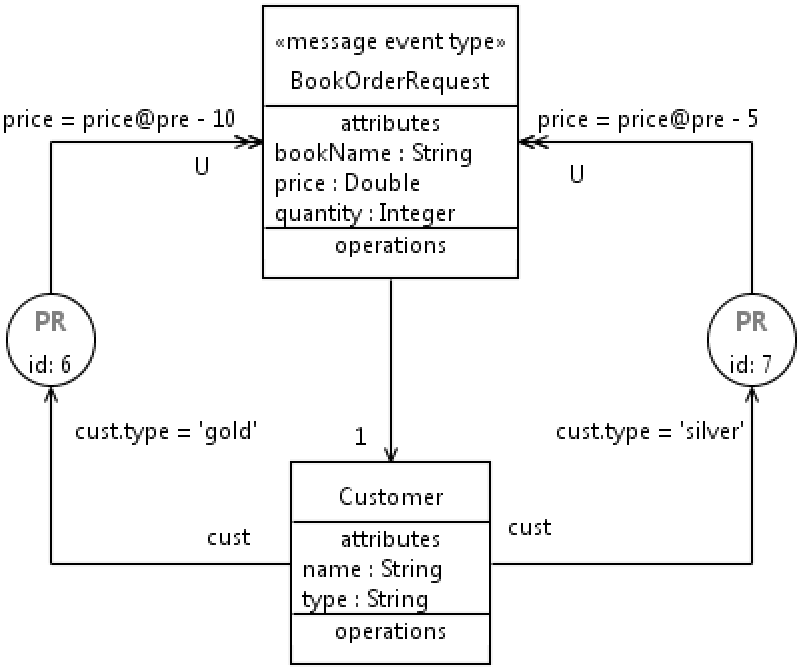


Figure 212. Reaction rules attached to the rule gateway (R_3) in Figure 210

When the *BookOrderRequest* message is received by the bookstore, we come to our fourth variablity point, *Bookstore update of a book stock quantity*. Here, we use two tasks, the *Invoke binding service* and the *Assign discount* task, in order to assign appropriate discount rules to the rule gateway (R_4). These rules are shown in Figure 213. We should note that we used these tasks and rules in the *Calculate discount* subprocess, too. Then, these rules are used to update the price in the *BookOrderRequest*, i.e., to decrease it by 10 money units if the customer is 'gold' (type) or to decrease it by 5 money units if the customer is 'silver'. When the rule is selected and attached to the rule gateway (R_4), the sequence flow invokes the rule gateway (R_4).



a) b)

Figure 213. Dynamic production rules attached to the rule gateway (R_4) in Figure 210

However, we have one more rule attached to the rule gateway (R_4). We show this rule in Figure 214. We used this rule to decrease the quantity in stock of the requested book, and then to invoke the *Send book* task in order to send the *BookOrderResponse* message with receipt to the customer. This pattern (the *Decision node to business rule binding* pattern), enabled us not only to dynamically choose the appropriate rule that is used to give a discount, but also to allow a business users to either change or add new rules during the process execution. This is the main advantage of rBPMN here regarding the standard BPMN.

When the book is received by the customer by using the *Receive book* task, we came to the fifth variablity point, *Customer decision to return a book*. As already stated, we do not model this point by using rules, as this decision implies a human decision, so it should not be automated in the process.

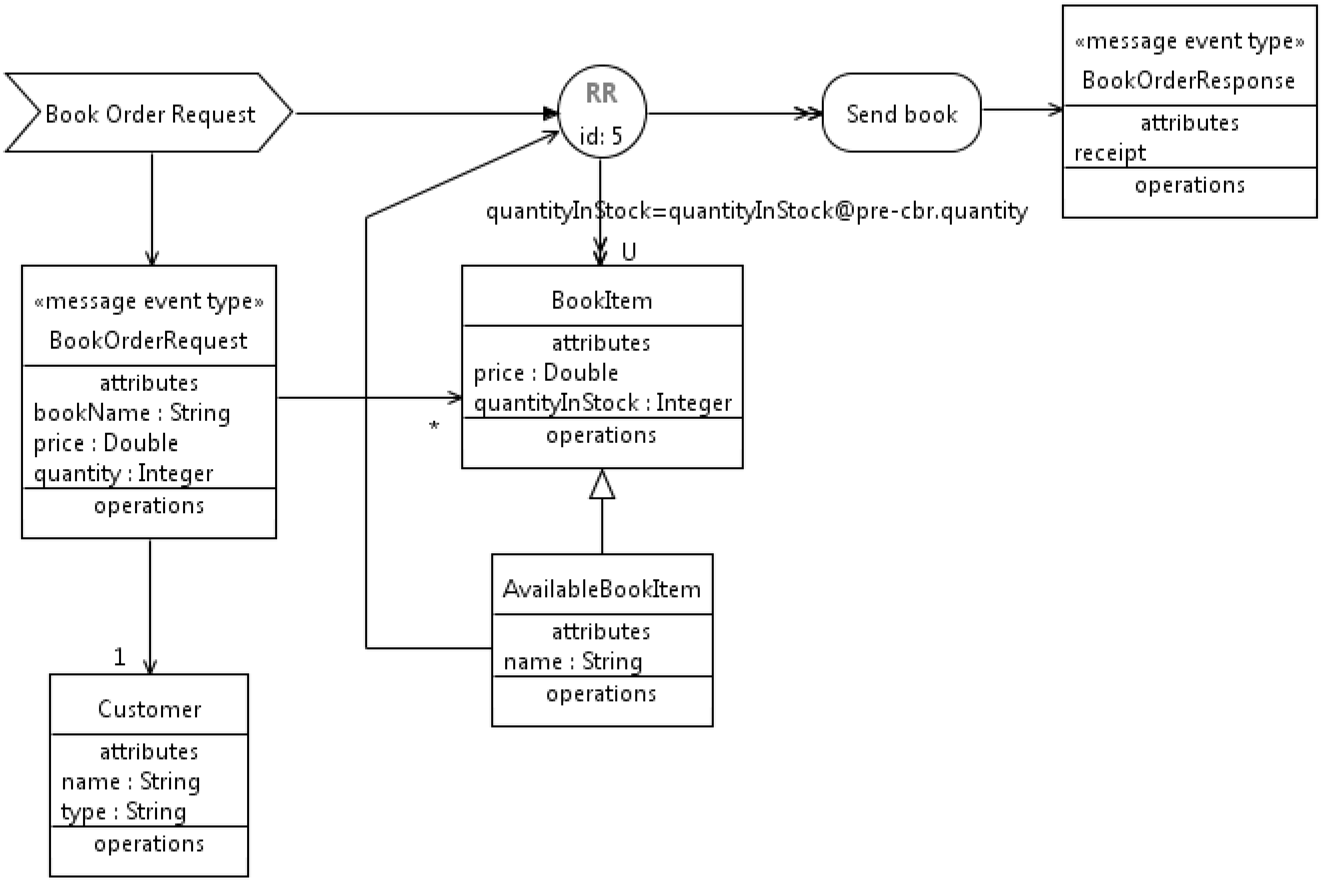


Figure 214. Static reaction rule attached to the rule gateway (R_4) in Figure 210

If the customer decides to return the book, the *Book return request* task is invoked in order to send the *CustomerBookReturnRequest* message to the bookstore, and this is our sixth variability point, the *Bookstore rollback payment decision*. Here, the bookstore need to decide whether to send a new book or to rollback the payment. This decision depends on the customer request, and we implemented this decision by using two reaction rules (see Figure 215), so that we can choose appropriate subprocess that is outgoing from the rule gateway (R_5). If the user wants to abort the order (i.e., he does not want a new copy of a book, or a correct book if a wrong one is delivered), the *abortOrder* attribute of the *CutomerBookReturnRequest* message is set to true, and the rule with the positive condition is enabled; otherwise, the rule with the negated condition is enabled. In the first case, the *Rollback payment* subprocess is selected, and in the second case the *Send book* subprocess is selected. This enables for selecting dynamically the subprocess, as we can add more conditions to the rule gateway.

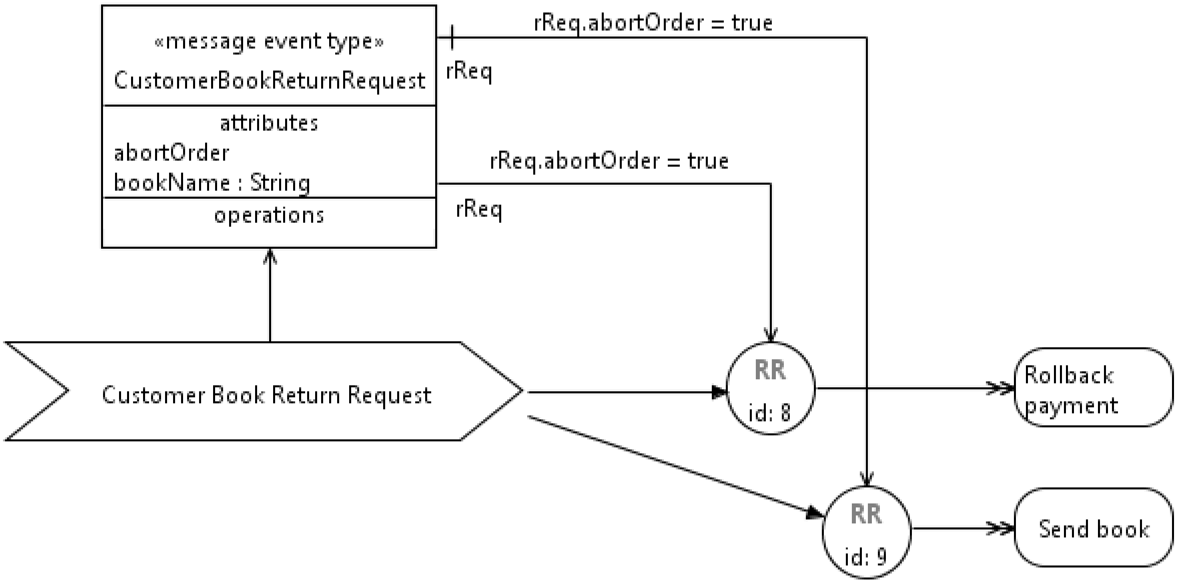


Figure 215. Reaction rules attached to the rule gateway (R_5) in Figure 210

If the customer decided to abort the order, the process flow goes on to the *Money receipt* task from the rule gateway (R_6), which is enabled when the message is sent from the bookstore (by using the *Send money back* task). The rule gateway (R_6) also has two reaction rules attached and uses the same condition as the rules shown in Figure 215, with a diffrence that the rule with the positive condition invokes the *Money receipt* task (i.e., enables the task and wait for a messsage from the bookstore), and the rule with the negated condition invokes the *Receive book* task. As these rules are similar to the one show in Figure 215m we omit their definition from the text of this thesis.

5.4. rBPMN Language Implementation: the rBPMN Editor

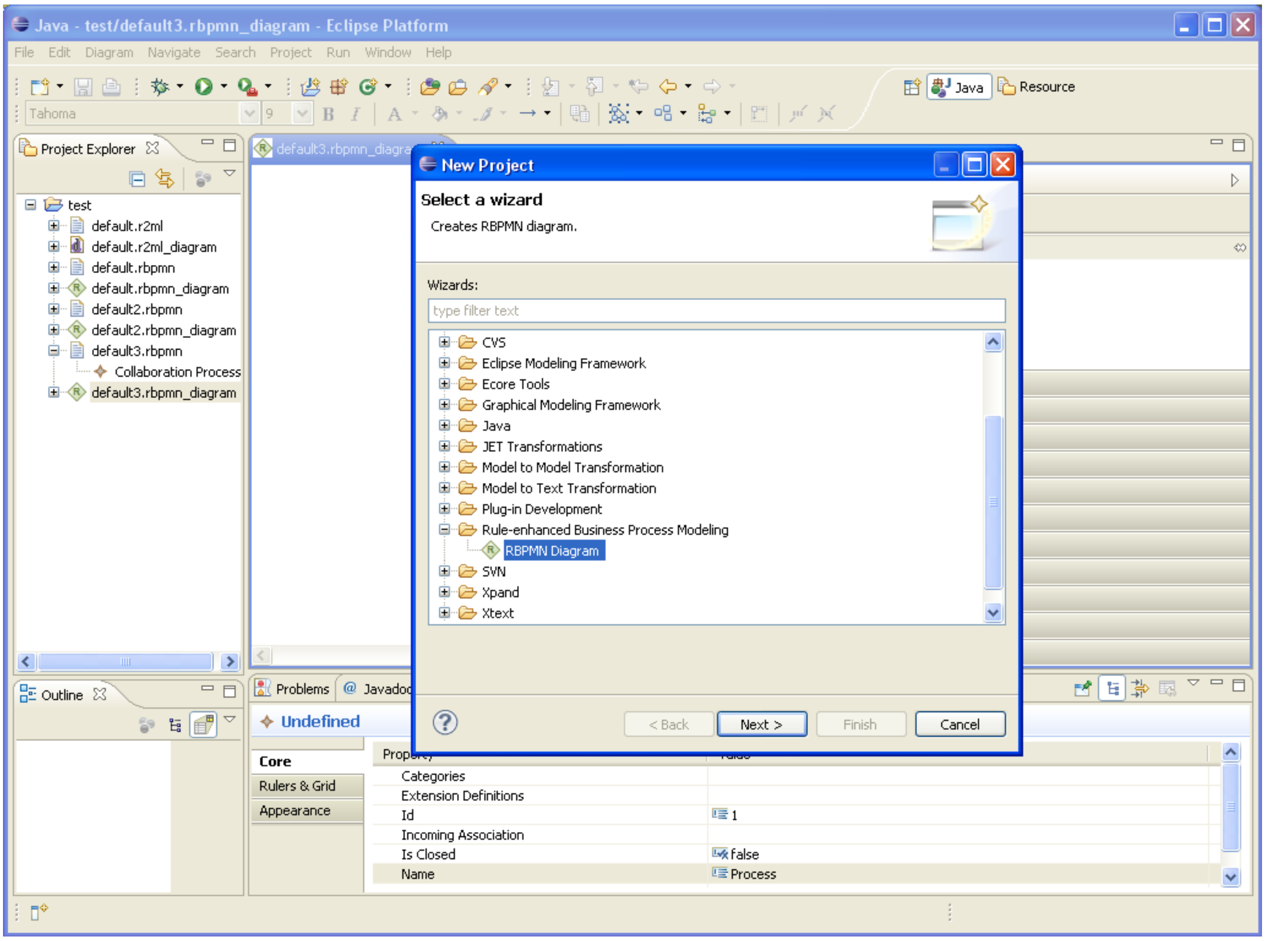
Along with the rBPMN language, we also developed an rBPMN graphical editor as a proof of concept for our conceptual contributions presented in this thesis. The editor is built by using the Eclipse GMF framework [29], which allows for producing a set of Eclipse plug-ins. We developed the editor starting from the rBPMN metamodel (see section 3.2) in the Ecore format.

We should note here that we changed an actual implementation of the conceptual rBPMN metamodel in the rBPMN editor, due to GMF constraints. Regarding the triggering and triggered concepts from Figure 114, we inherited FlowNow from *AtomicEventExpression*, so that triggering and triggered events can be any activity, event or a gateway, because we didn't want to introduce new graphical elements in diagrams.

In addition, a FlowNode is introduced as an attribute of the R2ML's *ActionEventExpression* class, so that we can have traceability between BPMN2 activities and R2ML actions. The R2ML's *AtomicEventExpression* is inherited from BPMN2 *ItemDefinition*, so that we can choose a *AtomicEventExpression* message in the *structureRef* attribute of the BPMN2 *Message* element.

We will describe the usage of the rBPMN editor in the modeling of a rBPMN processes orchestration from section 5.1, in the following subsequent steps.

Step 1. Creating rBPMN diagram. When we run the generated editor as an Eclipse application, in the new Project Wizard (see Figure 216), we added a Rule-enhanced Business Process Modeling category, with RBPMN Diagram wizard. If the rBPMN Diagram wizard is choosed, the project and two models are created: a model file and a diagram file (both ecore), and the rBPMN process diagram is opened in the middle window of the application, denoted by the name entered in the wizard.



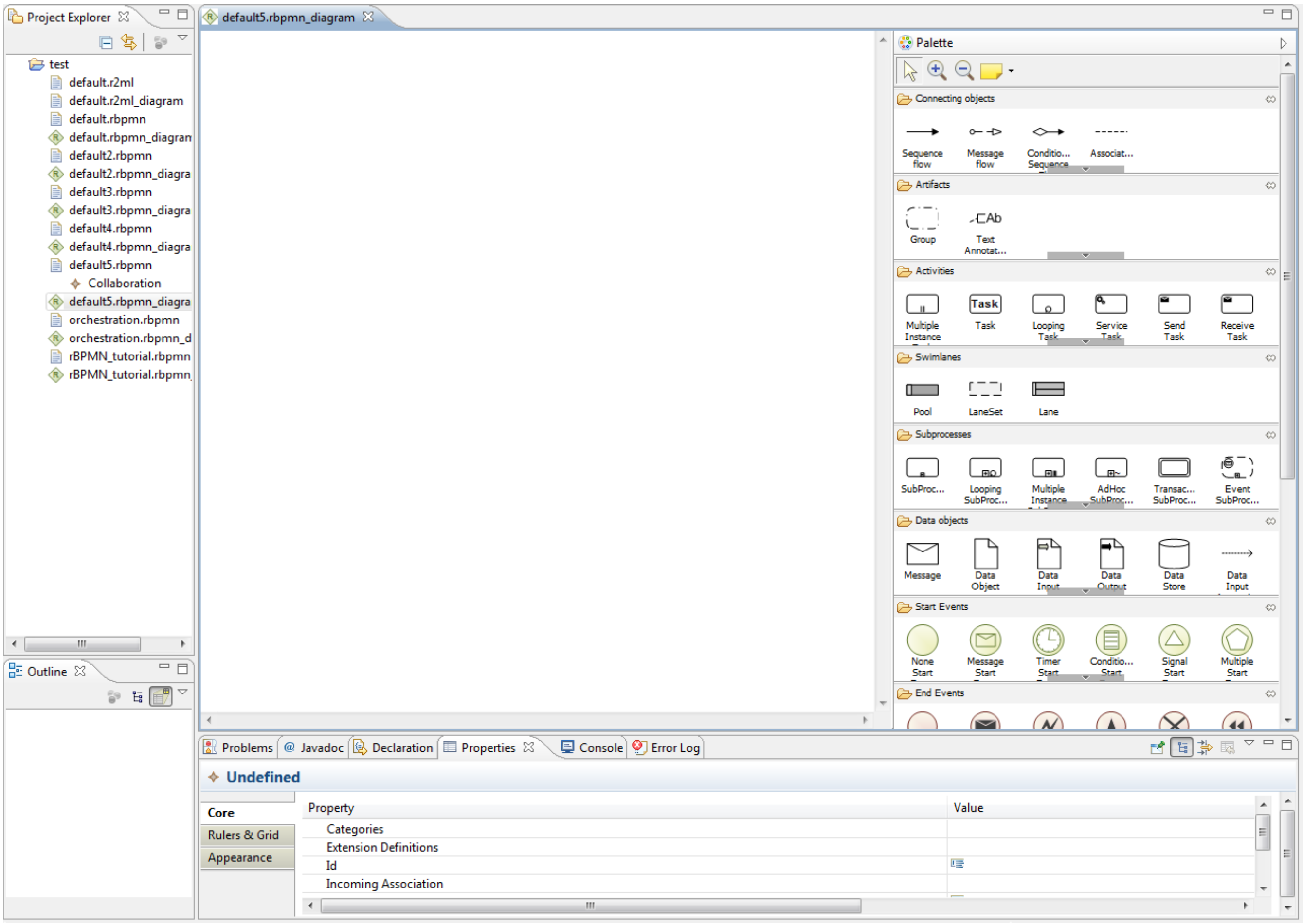
Select a wizard

Creates RBPMN diagram.

Property	Value
Categories	
Extension Definitions	
Id	1
Incoming Association	
Is Closed	false
Name	Process

Figure 216. New rBPMN Diagram wizard

When the diagram is instantiated, the application looks like in the Figure 217. In the center we have an actual model in rBPMN, which is drawn. On the bottom, we have Properties for the selected element (ecore property of an element) and on the right side we have the Palette with the tools for creating different rBPMN elements. The rBPMN editor represents a fully functional, BPMN2 editor, where rBPMN extensions are added in the Rules pallete group.



Property	Value
Categories	
Extension Definitions	
Id	
Incoming Association	

Figure 217. An empty instance of a rBPMN diagram in the rBPMN editor

Step 2. Process design. In order to create new elements in the editor, a user can select them from the Palette tab (see Figure 127). When the element is selected, a user just needs to click on the diagram to create that element. Such an element is created in the model file but also in the diagram file.

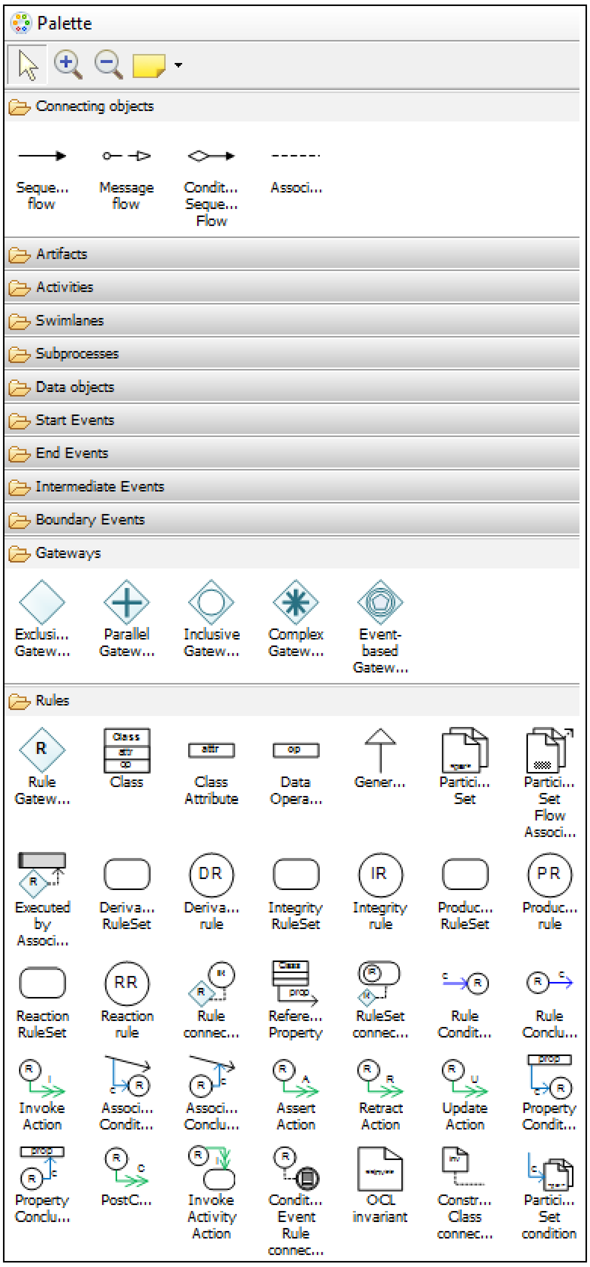
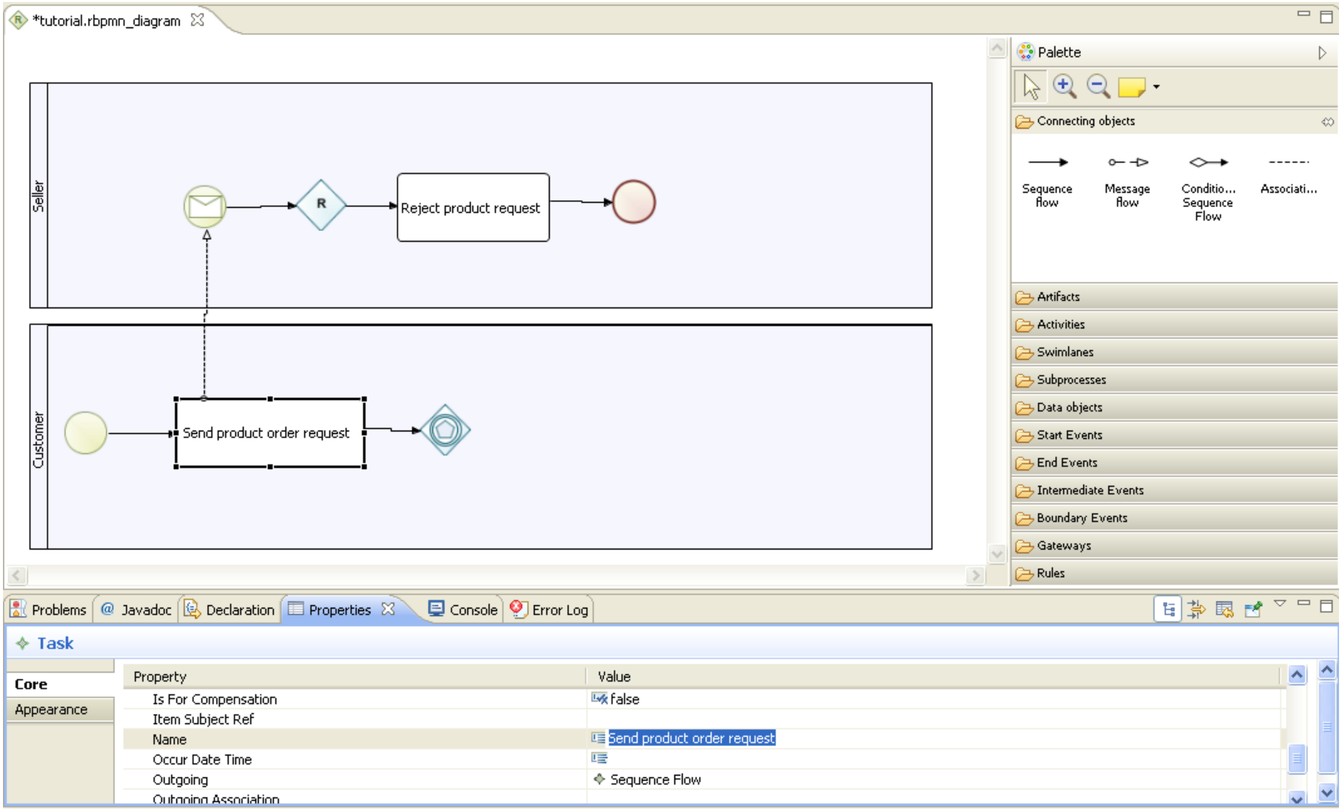


Figure 218. Palette in the rBPMN editor

In the rBPMN language, similar to the BPMN language, we have pools as main elements, and a start event as the first element in a pool. We can start modeling by placing a pool on a diagram (Seller, Customer), and then we can add an empty start event in the pool (see Figure 219). Each element can be moved in all of the directions or resized. When an element is selected, by typing a text, it name can be changed. In addition, in the Properties window can be used to change basic attributes of any graphical element(see Figure 219). Next, we can draw connections between elements, i.e., a sequence flow between events and tasks in a one pool, and a message flow between tasks or events in the separate pools. In this way we can draw all of the rBPMN elements, including gateways, subprocesses, artefacts, etc.



Property	Value
Is For Compensation	false
Item Subject Ref	
Name	Send product order request
Occur Date Time	
Outgoing	Sequence Flow
Outgoing Association	

Figure 219. Drawing basic process elements in the rBPMN editor

Along with basic BPMN2 elements we can add rule-based elements in a rBPMN process too, by using the tools from the Rules tab. These elements are vocabulary elements (class, their attributes and operations, and a generalization), different rule sets and concrete rules (derivation, integrity, production and reaction), as well as the rule gateway. The Rules tab includes elements for the rule gateway and rules connection (Rule connection), and also rule conditions, conclusions and actions (assert, retract, update and invoke). Near a rule gateway in a process, the user needs to add a particular ruleset (reaction, production, etc.) and a rule in the ruleset, with a connection from the rule to the rule gateway. All of the rule elements can be drawn in the rBPMN editor, however we recommend defining only a basic rule elements in the process editor and the using a more complete Rule editor for more precise rule definition (see step 3 and 4).

When the rBPMN process diagram is finished, we get the process as shown in the Figure 220.

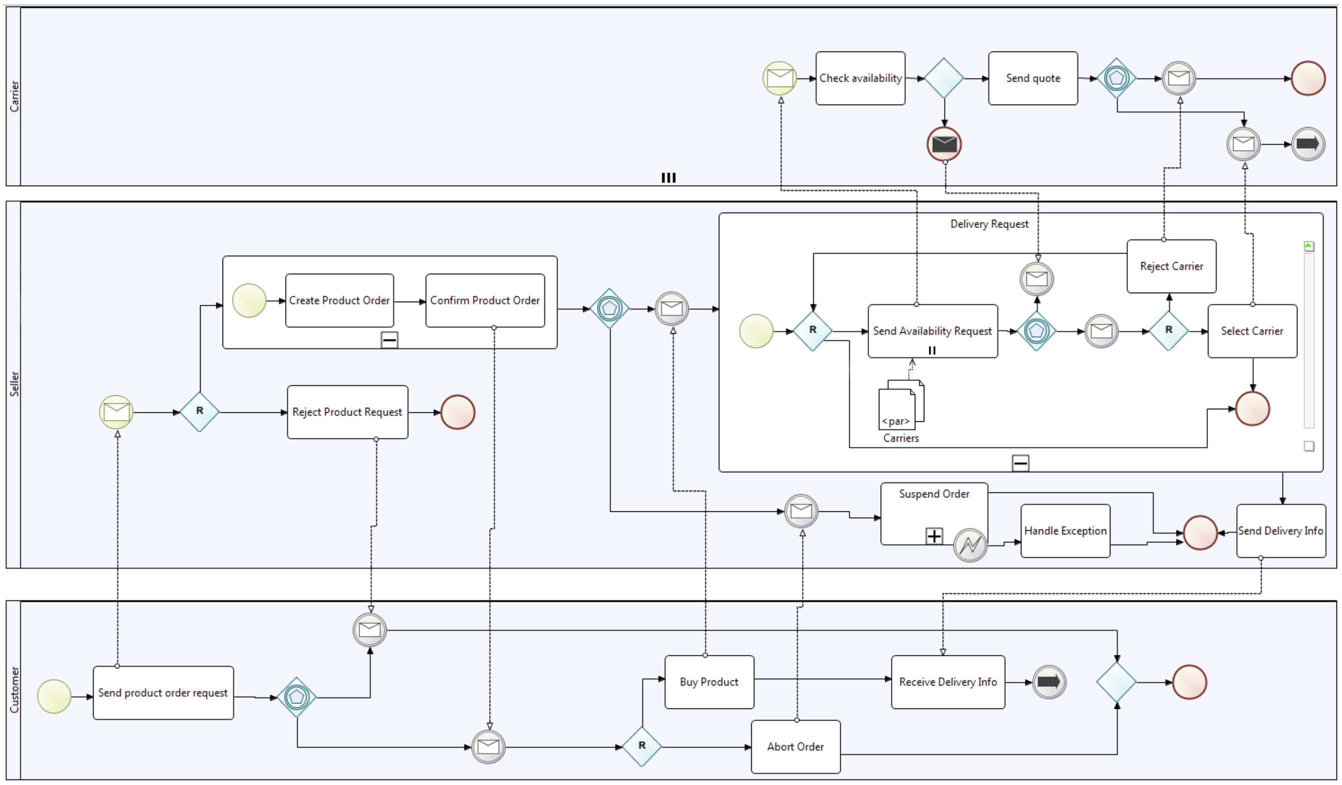


Figure 220. The rBPMN diagram from Figure 200 drawn in the rBPMN editor

When the process diagram is finished, we can also add data objects from the Palette or a Participant set from the Rules tab in order to store reference to multiple participants in a process (multi-instance pools), as shown in Figure 220 for the *Carriers* participant set.

Step 3. Data design. When a plain business process is drawn by using BPMN2 elements, we need to integrate vocabulary (data) elements in such process, as they are also used in rules definition. A vocabulary is a part of the ruleset and we first need to add a new ruleset to the process diagram (see Figure 221). In the ruleset we can add a rule of the ruleset type and connect the rule with the rule gateway by using the Rule connection. When the ruleset is created and a rule is added to it, by double-clicking the ruleset the Rule editor opens.

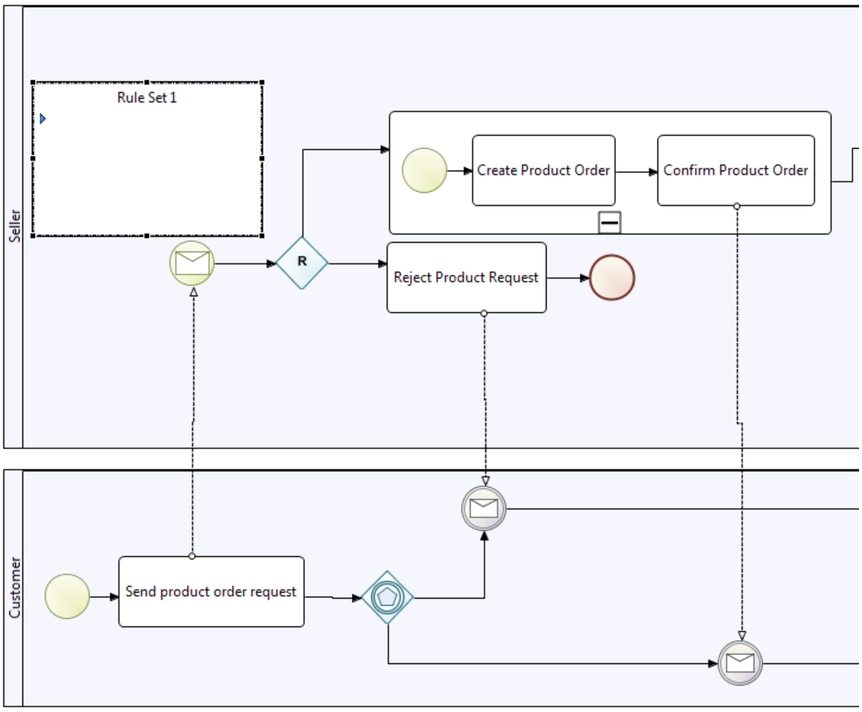


Figure 221. Rule Set 1 in the rBPMN process diagram

In the Rule editor, we can see the Pallete of tools for the particular ruleset type (see Figure 222). Here we can draw vocabulary and rule elements in a rule diagram.

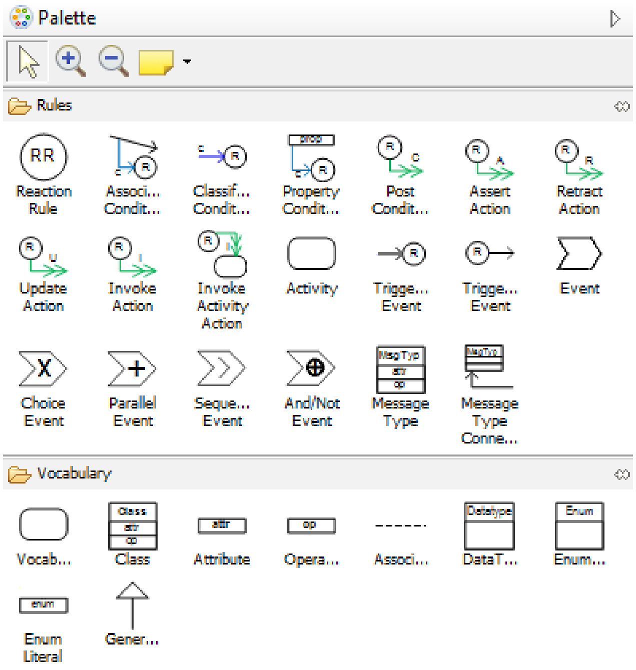
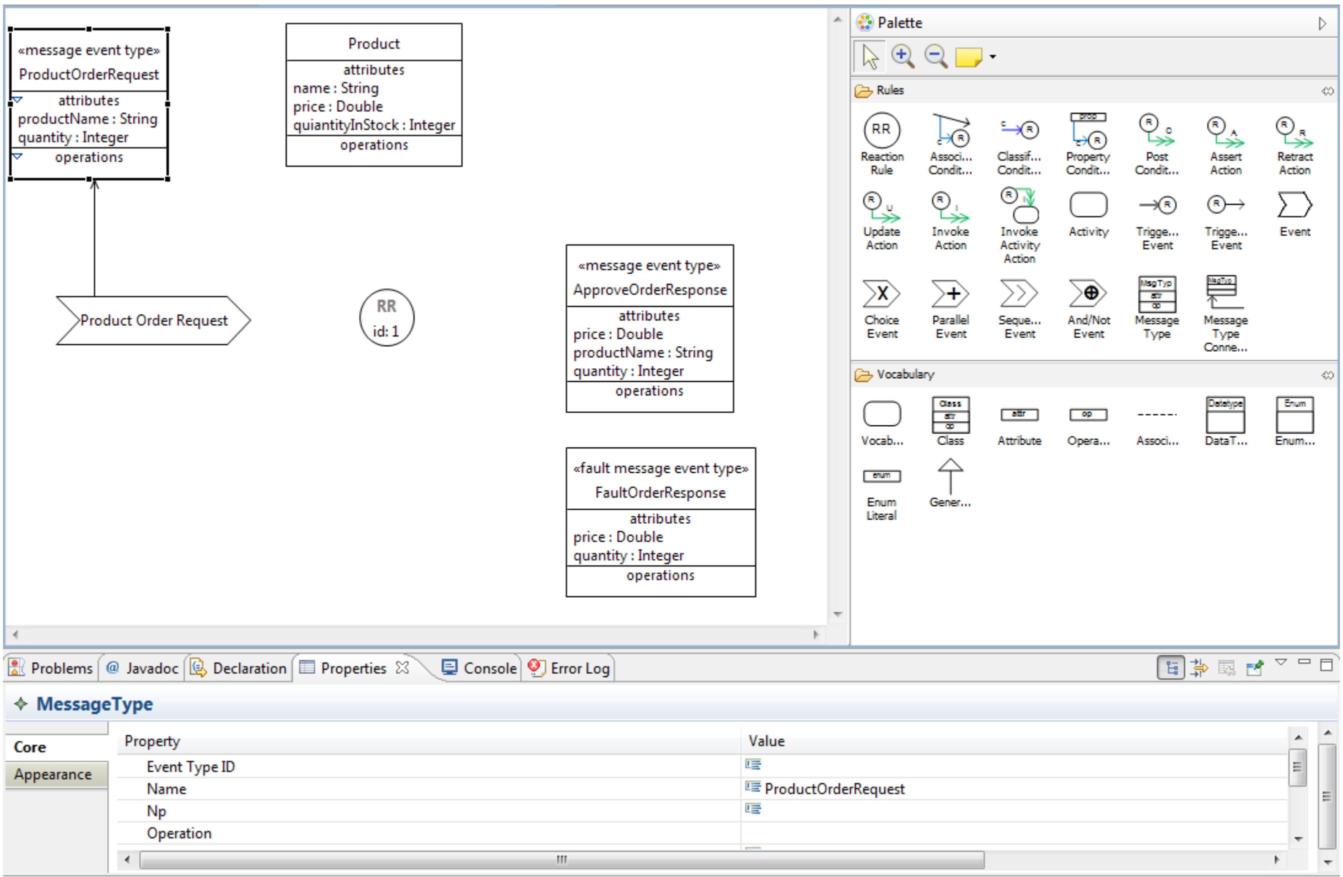


Figure 222. The Rules editor Palette

First, we need to add vocabulary elements, which are then used in a rule definition. In order to do this, we need to add a rule from the Palette, and then all of the needed Classes for rule conditions and (fault) message event types, used in rule triggering and triggered event definitions or returned messages from an activity that is invoked by a rule action. These elements are added in a diagram by choosing the Class and Message Type elements in the Palette, respectively, and placing them to a diagram. Then, by chosing the Attribute or Operation Palette element, we can add needed attributes and operation in a

Class or Message Type. Each Message Type has a “message event type” stereotype, which can be changed in the Properties windows for a Message Type. In Figure 223 we have defined all of the needed vocabulary elements for the rule diagram shown in Figure 201, with the Product Order Request Event. The Message Type Connection is then used to connect the Product Order Request Event to the ProductOrderRequest? Message Type.



Property	Value
Event Type ID	
Name	ProductOrderRequest
Np	
Operation	

Figure 223. Modeling a rule diagram in the rBPMN Rule ditor

Step 4. Rule design. In this step, we connect the rules with conditions and events. When an Event is placed in a diagram, in this case, the Product Order Request and connected to the ProductOrderRequest message event type, we use the Triggering event Palette tool to connect them. Next, we use the Classification condition in order to connect the rule (RR id:1) with the Product class and to define the condition on this Classification condition. We do this by clicking on the Classification condition, in the Palette, then clicking on the rule (RR id:1) and release it on the Product class (see Figure 226). The actual condition can then be inserted in the diagram in the place of the condition line (or in the filter attribute in the Properties windows for the connection). When the condition is defined, we need to add activities in a sequential order (Create Order and Confirm Order), which are located in the main process after the rule gateway. These activities are invoked by the rule (RR id:1) if the condition on the rule is satisified (name = prVar.productName and quantityInStock > prVar.Quantity). In order to define sequential activities, we need to choose the Sequential Event from the Palette and put it in the diagram. After that, we can add activities (the Activity tool from the Palette) in the Sequential Event. When all of the activities are placed, we just need to connect the rule with the first invoked activity (Create Order), by using the Invoke action from the Palette, and also the ApproveOrderResponse message type with the Confirm Order activity by using the Message Type Connection. In addition, we have an Update action from the rule (RR id:1) to the Product class, used to decrease the quantityInStock attribute (needed in the case when the book order is created). In order to define this action in the diagram, we need to click to the Update action in the Palette and then on the rule and release it on the Product class. The update variable can be inserted in the Properties window for the Update action, by changing the Class Variable text attribute.

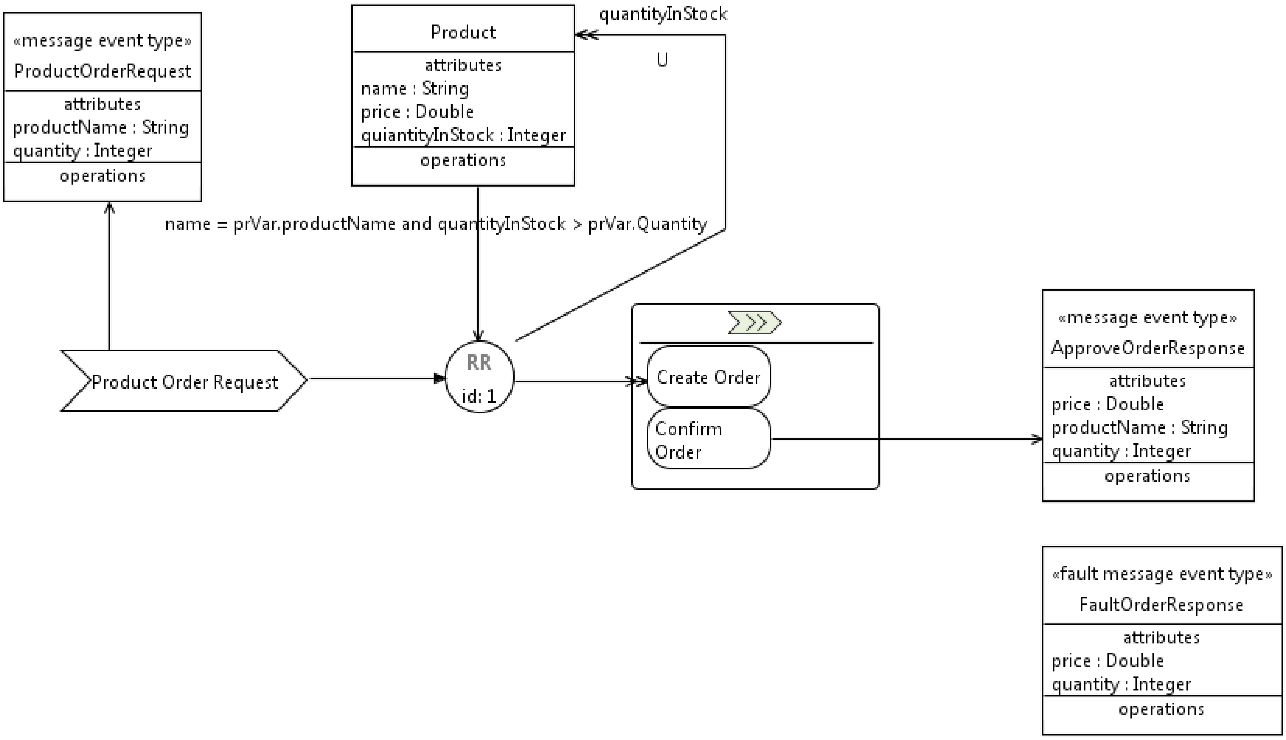


Figure 224. Modeling the first Reaction rule from Figure 201

After defining the first rule, by using the same elements from the Palette we need to define the second reaction rule (RR id:2), define a negated condition of the first rule (Classification condition), and the Reject Product Request activity. Then, we just need to connect the rule (RR id:2) with the Product Order Request event (by using the Triggering event tool) and the Invoke action to the Reject Product Request activity. Lastly, we also need to connect the Reject Product Request activity with the FaultOrderResponse message, which is returned if the RR id:2 is activated, by using the Message Type Connection.

6. Analysis of the proposed solution

In this chapter, we analyze the proposed rule-based language. In order to evaluate the rBPMN language, we compared it to the existing process modeling languages for interaction modeling, control flow modeling and regarding a way of improving the agility of a process. We conducted this evaluation by using service interaction patterns (see Section 4.2) for interaction modeling, control flow patterns (see Section 4.3) for control flow modeling and agility patterns (see Section 4.4) for the agility of a process.

6.1. Comparison of Business Process Modeling Languges for Basic Control Flow Patterns

In this section, we compare rBPMN the languages for workflow and business process modelling. If a language directly supports a pattern through one of its constructs, it is rated +. If a pattern is not directly supported, but can be "mimicked", it is marked with +/-. Any solution which results in "spaghetti diagrams" or coding to compensate sematics of a pattern is considered as giving no support and is marked with -. Table XI includes the following langauges: Extended AORML [128], XML Process Definition Language (XPDL) [149], UML Activity Diagrams [96], BPEL4WS (Business Process Execution Language for Web Services) [49], WS-CDL (Web Services Choreography Description Language) [55], and BPMN 1.2 (Business Process Modeling Notation) [88]. We used the information about WS-CDL and BPEL workflow pattern support from [26], for XPDL, UML and AORML from [128], and for BPMN from [148].

Table XI. Comparison of workflow and business process modeling standards

Pattern group	**Pattern**	**Business process modeling languages**						
		WS-CDL	XPDL	UML	BPEL	BPMN	AORML	**rBPMN**
Basic control-flow patterns	Sequence	+	+	+	+	+	+	+
	Parallel Split	+	+	+	+	+	+	+
	Synchronization	+	+	+	+	+	+	+
	Exclusive Choice	+	+	+	+	+	+	+
	Simple Merge	+	+	+	+	+	+	+
Advanced branching and synchronization patterns	Multi Choice	+	+	-	+	-	+	+
	Multi Merge	+/-	-	-	-	+/-	+	+
	Discriminator	-	-	-	-	-	+/-	+
	Synchronizing Merge	+	+	-	+	+	-	+
Structural patterns	Arbitrary Cycles	-	+	+	-	+	+	+
	Implicit Termination	+	+	+	+	+	+	+
Multiple Instances patterns	MI without synchronization	+	+	+	+	+	+	+
	MI with a Priori Design Time Knowledge	+	+	+	+	+	+	+
	MI with a Priori Runtime Knowledge	-	-	+	-	-	+	+

	MI without a Priori Runtime Knowledge	-	-	-	-	-	+	+
State-based patterns	Deferred Choice	+	-	+	+	+	+	+
	Interleaved Parallel Routing	-	-	-	+/-	+/-	-	+/-
	Milestone	+	-	-	-	-	-	+
Cancellation patterns	Cancel Activity	+	-	+	+	+	+	+
	Cancel Case	+	-	+	+	+	+	+

From Table XI, we can see that all the languages support all the patterns in the first group – *Basic control flow patterns*. In this group, we introduced rules combined with a rule gateway in order to more define precisely conditions under which an activity or sequence of activities will be invoked. By using rules in this group, we enabled to define but also to change those conditions at run-time or at design-time, also. This introduces a new flexibility to a process, because a process now must not be static any more.

The usage of rules in the second group of *Advanced branching and synchronization patterns*, is more obvious. For *Multiple Choice* pattern one of the important criterions for support of this pattern is that is needed to define logical condition(s) on outgoing branches, which can be changed in run-time. This is exactly the advantage of our solution, because reaction rules are connected to a vocabulary that can be changed at run-time, or at design-time. Additionally, we have supported a *Discriminator* pattern by using a reaction rule attached to the rule gateway to precisely define when subsequent activities should be invoked. This pattern is not fully supported in the standard BPMN, because the meaning of the Complex-join gateway, i.e., its *IncomingCondition* expression, is not clearly supported to be used for this pattern.

In the third group, *Structural patterns*, we used reaction rules to more precisely define loops' entry and exit points, where conditions on these rules can be changed dynamically. We should also mention that our solution enables us to make an ordering in activity invocation by using a shared vocabulary, as described in *Implicit Termination* pattern. Regarding the *Arbitrary Cycles* pattern, rBPMN business process diagrams do not impose any restrictions on the structure of cycles, a business process model in our may have multiple entry and exit points which are represented as reaction rules.

In *Multiple Instances patterns* group, our solution is crucial especially for patterns with and without a priori runtime knowledge. In this group, we used reaction rules to dynamically change a number of required instances, as well as to define complex conditions on which an activity should be invoked. For Multiple Instance with apriori design time knowledge, we enabled to precisely define a number of created instances, and to define a condition when all of those instances have completed. This is the case as plain BPMN [88] does not have a language for defining conditions/expressions.

In fifth group, *State-based patterns*, by using rules and their vocabulary we supported the *Milestone* pattern, which is not supported in plain BPMN, as it lacks of support for states. In this pattern, by using the same vocabulary between multiple rules, we enabled that some tasks could be invoked only when they are enabled by some precisely defined condition. The *Interleaved Parallel Routing pattern* is not supported completely in BPMN, nor in rBPMN, as core BPMN does not support interleaving groups or sequences of tasks [116].

The final pattern group introduces two cancelling patterns. Our solution significantly improve *Cancel Case* pattern, because the usage of a rule gateway enabled us to cancel subprocess when activity fails, and also when a condition on a rule attached to the rule gateway is not satisfied.

6.2. Comparison of Languages Used for Modeling of Service Interaction Patterns

In this section, we present the results of the comparison of various languages for service interaction and business process modeling. If a language directly supports a pattern through one of its constructs, it is marked with +. If a pattern is not directly supported, but can be "mimicked" (e.g., see Dynamic routing pattern), it is marked with +/-. Any solution, which results in multiple diagrams or coding is considered as giving no support and is rated -. Table XII includes the following languages: WS-CDL (Web Services Choreography Description Language) [55], BPMN 1.2 (Business Process Modeling Notation) [88], Let's Dance [150], BPEL4Chor [23] and rBPMN. For evaluation of Let's Dance and BPMN, we relied on the work reported in [27]; for evaluation of WS-CDL and BPMN, we relied on the work from [26], for evaluation of BPEL4Chor we used the findings from [23] and for iBPMN and BPMN, we relied on the results from [25]. For evaluation rBPMN, we reported on the results from modeling the service interaction patterns shown in Section 4.2.

Table XII. Comparison of workflow and business process modeling languages for service interaction patterns

Pattern group	**Pattern**	**Language**					
		Let's Dance	BPMN	WS-CDL	BPEL4Chor	ext. BPMN	**rBPMN**
Single-transmission bilateral interaction patterns	Send	+	+	+	+	+	+
	Receive	+	+	+	+	+	+
	Send/Receive	+	+	+	+	+	+
Single-transmission multilateral interaction patterns	Racing incoming messages	+	+	+	+	+	+
	One-to-many send	+	-	+/-	+	+	+
	One-from-many receive	+	-	+	+	+	+
	One-to-many send/receive	+	-	+/-	+	+	+
Multi-transmission interaction patterns	Multi-responses	+	+	+	+	+	+
	Contingent requests	+/-	-	+/-	+	+/-	+
	Atomic multicast notification	-	-	-	-	-	-
Routing patterns	Request with referral	+	-	+	+	+	+
	Relayed request	+	-	+	+	+	+
	Dynamic routing	-	-	+/-	+/-	+/-	+/-

From Table XII, we can see that all languages support the first group of patterns, *single-transmission bilateral interaction patterns*, as this group of patterns is based on simple send/receive message interactions between two parties, which are supported in all the studied languages. The rules in this group of patterns, by leveraging a rule gateway, are used to define more precisely conditions under which the messages could be exchanged, as well as, for handling fault messages, as described for corresponding WSDL message exchange patterns in Section 4.1.As noted in the first group of patterns (see section 4.1.1), the Send pattern can be directly mapped into the Web Service Out-Only and Robust Out-only message exchange patterns (MEP), the Receive pattern with two MEPs: In-Only and Robust In-Only, and the Send/Receive pattern has two corresponding WSDL MEPs, Out-In and Out-Optional-In.

In the second group of patterns, *single-transmission multilateral interaction patterns*, which deal with multilateral interactions, the Racing incoming messages pattern is supported by all the languages. However, in rBPMN, we introduced a rule gateway to fulfill one of the design choices for this pattern [6], to define the ranking among competing received messages, so that only the message that fulfills the rule's condition can be consumed. The next three patterns, including One-to-many send, One-from-many receive, One-to-many send/receive, are not supported in BPMN 1.2, as reported in [22] that "BPMN cannot specify to which participant a message is sent, only the participant type is defined". In addition, the One-to-many send and One-to-many send/receive patterns are only partially supported in WS-CDL, as the number of message recipients is not known at design-time [26]. Thus, as we cannot know how to distinct between different participants of the same type (multi-instance Pool's) in BPMN 1.2, we introduced participant sets and reference passing in rBPMN (as in extended BPMN), along with the multiplicity of participants (Pool's). These participant sets are similar to reference sets and references in extended BPMN and iBPMN, which are used to reference particular participants in patterns that have multiple participants of the same type involved. The importance of using rules in this group of patterns, especially for the One from Many Receive and One to Many Send/Receive patterns, is in defining the stop and success conditions. These conditions are precisely defined in a declarative way, by using a rule gateway to decide whether the interactions are complete or whether they are successful completed or not. We must say that all the other languages but rBPMN can not define such conditions, even though they are required in these patterns. Also, the rules are used to define conditions under which tasks could be invoked.

In the third group of patterns, *Multi-transmission interaction patterns*, where one participant sends (receives) more than one message to (from) the same logical participant, the Atomic multicast notification patterns is not supported by any of the languages presented. None of these languages supports distributed transactions needed for implementation of this pattern. The Multi-responses is supported in all the languages. However, the other languages did not show how the stop condition or fault message return should be implemented. On the other hand, in rBPMN, returning a fault message from an involved participant in an interaction and defining a stop condition as an exit condition in a process-based while loop is encoded by using the rule gateway. The usage of rules enabled us to more precisely define the abovementioned patterns, i.e., their design choices as described in [6]. The Contingent requests pattern is just partly supported in Let's Dance, WS-CDL and extended BPMN, as these languages cannot accept messages from previous requests that failed due to a timeout [25][26][27]. We have supported this issue by using the reaction rule attached to the rule gateway, to define the condition when such responses should be accepted.

In the fourth group, *Routing interaction patterns*, the first two patterns (i.e., the Request with referral and Relayed request patterns) are supported in all the languages except BPMN, as BPMN does not support link passing mobility. We supported this issue in rBPMN by passing participant references among participants. We used rule gateways in these two patterns to support decisions on invoking tasks and on the decision whether the participant should return the fault message or not, which is not implemented in other languages than rBPMN. The Dynamic routing pattern is only partly supported in WS-CDL and rBPMN. In rBPMN, we supported the dynamic routing condition by using the rule gateway on data contained in the original request or in one of the intermediate steps. It is only partly supported as it define that a participant can insert new or delete existing interactions in the choreography at runtime, which is not currently supported in rBPMN nor in WS-CDL.

We should also mention, that only extended BPMN (iBPMN) and rBPMN support service interaction patterns through interaction models, along with behavior interconnected models, as we consider interaction models important for defining process and service choreographies.

6.3. Analysis of rBPMN usage for modeling agility patterns

In this section we show comparisons of original support for agility patterns presented in [42] [43] and their support in rBPMN, as shown in previous sections. We analyze three pattern groups in usage of different rules types for their realization. Each pattern is realized by using one or more rule types.

Table XIII. Summary of different rule types usage in agility patterns

		Original agility patterns [42]			**rBPMN**			
		Derivation rule	Constraint	Process rule	Derivation rule	Integrity rule	Production rule	Reaction rule
Control flow decisions	Decision Logic Abstraction pattern	+			+		+	
	Decision Node to Business Rule Binding pattern	+			+		+	
	Decision with flexible input data pattern	+			+		+	
	Decision flexible output pattern	+		+			+	
Data constraints	Constraints at predefined checkpoint pattern		+			+		
	Constraints at multiple checkpoints pattern	+	+			+		
	Constraints enforced by external Data Context pattern		+			+		+
Dynamic business process composition	Business rule-based subprocess selection pattern			+			+	
	Business Rule based Process Composition pattern	+		+	+		+	

In the first pattern group, called *Control flow decisions*, we supported the first pattern in the group (*Decision Logic Abstraction* pattern) by using derivation rules attached to a rule gateway, as this pattern implies forking based on a decision. In this group of patterns, we proposed how rule gateways can be used in the case of multiple choices from a one branching point. The main improvement we gain here is that by externalizing decision logic in rules, we allowed updating decision logic in runtime, without redeploying the business process. In addition, by using graphical representation for representing business rules we do not lose holistic view onto a process, as reported in [42]. For the second pattern in this group, *Decision Node to Business Rule Binding pattern*, the mapping service is introduced to dynamically attach a business rule to a decision node (rule gateway). For this mapping, we propose using business rules, such as production rules. In the *Decision with flexible input data pattern*, we have shown how input data for (not only) derivation rules can be passed during the runtime, while in the case of the Decision flexible output pattern we have shown how production rules can be used to dynamically assign activities to decision output branches, that should be invoked. This makes business process flexible, especially in the case of exceptional situations.

In the second group of patterns, we took into account three *Data constraints*. In patterns *Constraints at predefined checkpoint* and *Constraints at multiple checkpoints*, we used integrity rules to define constraints on a data model that should be checked before and after activities. Those constraints are defined on a data model, as they usually need to validate data. The usage of rules in these patterns enabled to update these constraints or add new constraints dynamically, durig runtime. In the third pattern, *Constraints enforced by external Data Context* pattern, we externalized data context, so that it can be seen by an external process or applications. For realization of this pattern, we used a subprocess that will send a message or data to the External process, where the integrity rule will be used to check data on violation and to handle that violation. As constraints only depend on data, this enables us to add constraints without the changing process elements. In variant B of this process, we propose using reaction rules to handle violations, where we can use rule to change process data.

In the third group, we represented two *Dynamic business process composition* patterns. In the *Business rule-based subprocess selection* pattern, similarly to the *Decision with flexible output* pattern, we selected a subprocess that should be invoked instead of an activity, based on a production rule decision. The main advantage of this pattern is that the subprocess selection criteria can be changed during runtime. For the *Business Rule based Process Composition* pattern, we divided a business process into separate process fragments, where we dynamically invoked process fragment. By using derivation rules, we defined the process fragment execution ordering.

7. Conclusion

In this chapter, we show results that are achieved by the research presented in this thesis. We also comment on the potentials of practical use of developed meta-model and transformations and give some reflections on the future plan.

7.1. Achieved contributions

In this thesis we developed a methodology for creation of rule-driven business processes and Service Oriented Architectures. We developed the language called rBPMN based on abstract (metamodel) and concrete syntax of a business process language (BPMN) and rule language (R2ML). rBPMN is a language that provides a systematic integration of rules in business process modeling. In addition, we developed a software editor for modeling rule-based business processes (rBPMN editor). The proposed rBPMN language is the first solution that provides a systematic integration of a rule modeling language with a business process modeling language. This was done by leveraging the principles of MDE and by weaving the metamodels of the studied languages. Our analysis of the expressiveness of rBPMN for modeling service compositions showed that rBPMN increases the level of modeling support for different kinds of patterns. The improvement is especially evident for the patterns that are state based and assume the use of multiple instances of the same activities. This stems from the rBPMN ability to represent reaction and production rules defined over explicitly specified business vocabularies. Our analysis also demonstrated some important benefits even for the patterns which could have been modeled by other languages. In particular, the use of rules in rBPMN in modeling service compositions increases the level of precision and dynamism. The precision of models is achieved thanks to the use of logic-expressions that, for example, allow for defining more precise conditions under which a certain process should terminate. Similarly, due to their declarative nature, rules can be updated at runtime. Thus, service orchestrations can be maintained more dynamically. Moreover, we have also defined a detailed methodology which guides service engineers through the main modeling tasks related to service compositions. Detailed contributions of this research are given in the rest of the section.

In this thesis, we can emphasize the following conceptual contributions:

- Overview and analysis of disciplines which are relevant for the subject of the research, namely: the basic concepts of the Model Driven Engineering are defined, Service compositions and business processes are introduced, and an overview of basic rule languages is given;
- Overview of the design of a number of software and conceptual environments and modeling tools;
- Contributions to the general rule markup language for representing rules on the Web (concrete and abstract syntax), named REWERSE I1 Rule Markup Language (R2ML);
- Contributions to the general rule markup language (concrete and abstract syntax), named REWERSE I1 Rule Markup Language (R2ML);
- Extension of the basic concepts of the BPMN2 meta-model, i.e., design of the rBPMN language;
- Development of the integrated methodology for development of rule-driven SOAs;
- Improvement of determining the variability of a process during process design;
- Conceptual solution for integrating different types of rules and policies in a business processes;
- Design and development of rBPMN language, which enables:
  - generating of more complete service descriptions, which are fully based on business vocabulary types;
  - dynamic updating of parts of business process logic by means of four different types of rules;

  - capturing fragments of business logic in the form of business rules (easier to be understood and verified by business experts) and their visual positioning inside business processes;
  - connections between rule interchange with process modeling in order to show how parts of business logic can be shared between different business partners;
- Definition of message exchange by using rules, which enables: dynamic change of the flow in a process;
- Complete generation of Web service descriptions and other service execution languages (WS-BPEL and BPEL4Chor), as well as the architecture for integration of rules into BPEL;
- Conceptual traceability between orchestration and choreography models in rBPMN;
- Integration of rules in busines processes evaluated by different types of patterns: Message Exchange Patterns, Control Flow patterns, Service Interaction patterns and Agility patterns.
- Comparative analysis of the rBPMN language and other process languages;
- Comparative analysis of the proposed approach for integration of rules into processes with other existing solutions.

Practical contributions of this thesis are following:

- Analysis of existing technologies that can be used in developing rule and process languages;
- Comparative analysis of business rule approaches and languages;
- Design and development of an rBPMN meta-model and concrete graphical syntax;
- Implementation of the R2ML, BPMN2 and rBPMN metamodels;
- Implementation of a fully functional BPMN2 process editor;
- Implementation and adaptation of the rBPMN editor;
- Analysis and detail testing of recommended solution;
- Recommendation for further work directions based on presented research.

7.2. Usage domain

The integrated methodology presented in this thesis could be used not only for development of rule-based business processes in rBPMN, but also in other similar languages. By employing the methodology, one can also include semantic descriptions in business process models. In addition, our methodology does not imply the usage of concrete service, but the selection of concrete service based on QoS parameters.

An important advantage of our solution is that it is based on the BPMN, a de facto standard for modeling business processes, so rBPMN can be used by business experts. By using R2ML, it is possible to define rules in plain R2ML, but also to translate its rules to other rule languages (such as Drools or other rule languages for which we have defined translators [110]).

Our solution (rBPMN language and editor) can be used in modeling both, service orchestrations and choreographies, which we proven by using them in differn types of process patterns. However, we should note that rBPMN editor is an academic project, so we have not made it in a way such as a final product for the customers. It is on the start of its lifecycle, and we are planning to make it an open source in the future. However, it is obviuous that our solution have great potential, so we can except that it will be developed as a commercial solution.

The usage of our solution is possible everywhere where business processes are developed, and especially in a complex business process or as a part of existing Service Oriented Architectures, which are going to be make executable on a concrete platform.

7.3. Further work and research directions

Our further research includes work on conceptual solution for integration of rules into processes, as well as the practical implementation and extensiosn of rBPMN editor. Namelu, our plans may comprise:

- In the future work, we will provide a detailed implementation of the transformations between rBPMN and BPEL. Currently, there are several solutions that connect service orchestration engines with rule engines (e.g., those of ORACLE, ILOG, and JBOSS), but all those rule engines are only for production rules. However, we will also include other three types of rules.
- Another important task that we are working on is related to increasing the usability of our current tooling support for rBPMN.
- We will work on the evaluation of the proposed graphical notation of rBPBM by using the novel principles for visual languages.
- We will also work on mappings of a semi-structured English language into business rules of rBPMN, so that business analysts can easier understand and capture their rules. In this process, we will leverage the OMG’s Semantics for Business Vocabularies and Rules (SBVR) specification.
- Regarding the rBPMN editor implementations, we are going to integrate rBPMN editor with R2ML rule editor.
- In addition, we will implement concrete transformations between two types of rBPMN choreography models (i.e., interaction and interconnected), as well as onto rule-enhanced service composition engines and service languages by using metamodel principles and model transformations.

8. Literature

1. Alanen, M., Lilius, J., Porres, I., Tr*uscan, D., "Realizing a Model Driven Engineering Process*", TUCS Technical report No 565, 2003.

2. ADONIS tool, BoC-Group, 2009, [Online], Available: http://www.adonis-community.com/.

3. ARIS Toolset, IDS Scheer AG, 2009, [Online], Available: http://www.ids-scheer.com/en/ARIS/ARIS_Software/3730.html.

4. ATLAS Transformation Language (ATL). http://www.sciences.univ-nantes.fr/lina/atl, 2009.

5. Atkinson, C., Kühne, T., "*Model-Driven Development: A Metamodeling Foundation*", IEEE Software (spec. issue on Model-Driven Development), Vol. 20, No. 5, pp. 36.-41., September/October 2003.

6. Barros, A., Dumas, M., Hofstede, T., A., "Service Interaction Patterns: Towards a Reference Framework for Service-based Business Process Interconnection", Technical Report FIT-TR-2005-02, Faculty of Information Technology, Queensland University of Technology, Brisbane, Australia, March 2005.

7. Baumgarten, B., "Petri-Netze: Grundlagen und Anwendungen", BI-Wissen-schaftverlag, Mannheim, 1990.

8. Bellifemine, F., Poggi, A., Rimassa, G. Developing multi-agent systems with a FIPAcompliant agent framework. Software – Practice and Experience 31 (2001) 103-128.

9. Bézivin, J., "On the unification power of models", Software and System Modeling, vol. 4, no. 2, pp. 171–188, 2005.

10. Bézivin, J., Farcet, N., Jezequel, J-M, Langlois, B., Pollet, D., "*Reflective model driven engineering*", In Proceedings of UML2003, USA, 2003.

11. Billington, J., Christensen, S., van Hee, K. E., Kindler, E., Kummer, O., Petrucci, L., Post, R., Stehno, C., Weber, M., "The Petri Net Markup Language: Concepts. Technology, and Tools", In Proceedings of the 24th International Conference Applications and Theory of PetriNets (ICATPN 2003), Lecture Notes in Computer Science, 2003.

12. Boley, H., Tabes, S., Wagner, G., "Design rationale of RuleML: a markup language for semantic web rules", in Proc. Semantic Web Working Symposium (SWWS '01), Stanford University, 2001.

13. Bouché, P., "WS-CDL and Pi-Calculus", lecture at Business Process Management II, Hasso-Plattner-Institute, 2005.

14. Bradshaw, J. M, et al. (2004). Making agents acceptable to people. In *Int. Tech. for Inf. Analysis: Advances in Agents, Data Mining, and Statistical Learning*, Springer, 361-400.

15. Bry, F., Eckert, M., Patranjan, L., P., Romanenko, I., "Realizing Business Processes with ECA Rules: Benefits, Challenges, Limits", In Proceedings of 4th Workshop on Principles and Practice of Semantic Web Reasoning , Budva, Montenegro, 2006.

16. Burbeck, S., "The Tao of E-Business Services", [Online], Available, http://www-128.ibm.com/developerworks/library/ws-tao/, 2000.

17. Charfi, A., Mezini, M., "Hybrid web service composition: business processes meet business rules", In Proceedings of the 2nd international conference on Service oriented computing, New York, NY, USA, pp.30-38, 2004.

18. Cibrán, A., M., Verheecke, B., "Dynamic business rules for web service composition", In R. E. Filman, M. Haupt, and R. Hirschfeld (eds), Proc. of the Second Dynamic Aspects Workshop (DAW05), p. 13–18, 2005.

19. Conallen, J., "Building Web Application with UML", Addison-Wesley, 2nd Edition, 2002.

20. Damianou, N., Dulay, N., Lupu, E., Sloman, M., "The Ponder Policy Specification Language", in proceedings of Workshop on Policies for Distributed Systems and Networks (POLICY 2001). Springer-Verlag, LNCS 1995, Bristol, UK, 2001.

21. DaNAMiCS tool home page, 2009 [Online]. Available: http://www.cs.uct.ac.za/Research/DNA/DaNAMiCS/DaNAMiCS.html.

22. Decker, G., Barros, A., "Interaction Modeling using BPMN", In Proceedings of the 1st International Workshop on Collaborative Business Processes (CBP), LNCS 4928, pp. 206-217, Brisbane, Australia, September 2007. Springer Verlag.

23. Decker, G., Kopp, O., Leymann, F., Weske, M., "BPEL4Chor: Extending BPEL for Modeling Choreographies", in Proceedings of the IEEE 2007 International Conference on Web Services (ICWS), Salt Lake City, Utah, USA, July 2007. IEEE Computer Society.

24. Decker, G., Kopp, O., Leymann, F., Weske, M., "Interacting services: From specification to execution", Data Knowl. Eng., doi:10.1016/j.datak.2009.4.003, 2009.

25. Decker, G., Puhlmann, F., "Extending BPMN for Modeling Complex Choreographies", In R. Meersman and Z. Tari et al (Eds.): OTM 2007, Part I, LNCS 4803, pp. 24-40, Springer, 2007.

26. Decker, G., Zaha, M., J., "Pattern-based Evaluation of WS-CDL", Working paper, Faculty of IT, Queensland University of Technology, August 2006.

27. Decker, G., Zaha, M., J., Dumas, M., "Pattern-based Evaluation of Let's Dance", working paper, August 2006.

28. Eclipse foundation, Graphical Editing Framework (GEF), [Online], Available: http://www.eclipse.org/gef/.

29. Eclipse foundation, Graphical Modeling Framework (GMF), [Online], Available: http://www.eclipse.org/gmf.

30. Eijndhoven, T., van, Iacob, E., M., Ponisio, L., M., "Achieving Business Process Flexibility with Business Rules," *Enterprise Distributed Object Computing Conference, 2008. EDOC '08. 12th International IEEE,* vol., no., pp.95-104, 15-19 Sept. 2008.

31. Erl, T., Service-Oriented Architecture: Concepts, Technology, and Design, Prentice Hall PTR, 2005.

32. EU-Rent case study at the European Business Rules Conference web site, [Online], Available: http://www.businessrulesgroup.org/egsbrg.shtml.

33. Falkenberg, E., D., Hesse, W., Lindgreen, P., Nilsson, B., E., Han Oei, J., E., Rolland, C., Stamper, R., K., van Assche, F., J., M., Verrijn-Stuart, A., A., Voss, K., "*A framework of information system concepts*", The FRISCO report, 1998.

34. Favre, J., M., "Towards a Basic Theory to Model Model Driven Engineering", *In Proc. of the UML2004 Int. Workshop on Software Model Engineering (WISME 2004)*, Lisbon, Portugal, 2004. [Online]. Available: http://www-adele.imag.fr/~jmfavre/papers/TowardsABasicTheoryToModelModelDrivenEngineering.pdf.

35. Favre, J., M., "*Foundations of meta-pyramids: languages and metamodels – Episode II: story of Thotis the Babbon*", [Online]. Available at http://www-adele.imag.fr/~jmfavre/, 2004.

36. Gašević, D., "*Petri Net Ontology*", PhD thesis (in Serbian), Faculty of Organizational sciences, University of Belgrade, 2004.

37. Gašević, D., Đurić, D., Devedžić, V., "*Model Driven Architecture and Ontology Development*", Springer, 2006.

38. Ginsberg, A., "*RIF Overview*", W3C Recommendation, W3C Working Group Note, 22 June 2010, [Online], Available: http://www.w3.org/TR/rif-overview/.

39. Goedertier, S., Vanthienen, J., "Business Rules for Compliant Business Process Models", In Proceeding of the 9th International Conference on Business Information Systems (BIS 2006), pages 558-579, 2006.

40. Goerdetier, S., Haesen, R., Vanthienen, J., "Rule-based business process modeling and enactment", *Int. J. Business Process Integration and Management,* Vol. 3., No. 3., pp.194-207, 2008.

41. Goldszmidt, G., Osipov, C., "Make conposite busines services adaptable with points of variability, part1: Choosing the right implementation", IBM developerWorks, April 2007.

42. Graml, T., "Business Rules enable agile Business Process Management", Master thesis, Institute For Informatics, Der Ludwing-Maximilians-University Munchen, 2006.

43. Graml, T., Bracht, R., Spies, M., "Patterns of Business Rules to Enable Agile Business Processes", In proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007), Annapolis, USA, 2007, pp. 365-375.

44. Grosof, B. N., et. al. (2003) "Description Logic Programs: Combining Logic Programs with Description Logic". *In Proc. of 12th Int'l. WWW Conf.* 48-57.

45. Guirca, A., Lukichev, S., Wagner., G., "Modeling Web Services with URML", *In Proceedings of Workshop Semantics for Business Process Management,* 2006.

46. Horrocks, I., Patel-Schneider, F., P., Boley, H., Tabet, S., Grosof, B., Dean, M., "*SWRL: A Semantic Web Rule Language, Combining OWL and RuleML*", W3C Member Submission, 21 May, 2004.

47. Hughes, R.I.G., "*The Ising model, computer simulation, and universal physics*", In M. Morgan and M. Morrison (eds.), Models as mediators, Perspectives on natural and social science, Cambridge University Press, 1999.

48. Iacob, E., M., Jonkers, H., "Model-Driven Perspective on the Rule-Based Specification of Services", Enterprise Distributed Object Computing Conference, 2008. EDOC '08. 12th International IEEE , vol., no., pp.75-84, 15-19 Sept. 2008.

49. IBM Developerworks, “Business process execution language for web services, version 1.1.”, online: http://www-128.ibm.com/developerworks/library/specification/wsbpel/, 2003.

50. Intalio, “*STP BPMN Modeler: BPMN object model*”, http://www.eclipse.org/stp/bpmn/model/index.php, 2009.

51. Java rule Engine API, JSR 94, [Online], Available: http://jcp.org/aboutJava/communityprocess/final/jsr094/index.html, 2004.

52. Jensen, K., “Coloured Petri Nets: A High Level Language for Systems Design and Analysis”, Springer-Verlag, Berlin, 1992.

53. Kasi, V., Tang., X., “Design attributes and performance outcomes: A framework for comparing business processes”, In *Proceedings of the Eighth Annual Conference Interoperability for Enterprise Software and Applications*, Funchal, Portugal, March 2007, pp. 63-75.

54. Kappel, G., Rausch-Schott, S., Retschitzegger, W., "Coordination in workflow management systems - A rule-based approach," in *Proceedings of Coordination Technology for Collaborative Applications - Organizations, Processes, and Agents*, 1364 ed, W. Conen and G. Neumann, Eds.: Springer, 1998, pp. 99-119.

55. Kavantzas, N., Burdett, D., Ritzinger, G., Fletcher, T., Lafon, Y, “Web Services Choreography Description Language Version 1.0”, W3C Candidate Recommendation, November 2005.

56. Kaviani, N., Gašević, D., Milanović, M., Hatala, M., "Model-Driven Engineering of a General Policy Modeling Language", *IEEE Workshop on Policies for Distributed Systems and Networks (POLICY 2008)*, Palisades, NY, USA, 2008.

57. Kent, S., "*Model Driven Engineering*", In Proceedings of IFM2002, LNCS 2335, Springer, 2002.

58. Klint, P., Lämmel, R., Verhoef, C., "*Toward an engineering discipline for grammarware*", ACM TOSEM, 2005.

59. Knolmayer, G., Endl, R., Pfahrer, M., "Modeling Processes and Workflows by Business Rules," in Business Process Management, Models, Techniques, and Empirical Studies, W. M. P. van der Aalst, J. Desel, and A. Oberweis, Eds. London: Springer, 2000, pp. 16-29.

60. Krogstie, J., McBrien, P., Owens, R., Seltveit, H., A., "Information Systems Development Using a Combination of Process and Rule-Based Approaches," in Third Nordic Conference on Advanced Information Systems Engineering: LNCS, Springer-Verlag, 1991.

61. Kopp, O., Leymann, F., “Do We Need Internal Behavior in Choreography Models?”, In: Kopp, Oliver (ed.); Lohmann, Niels (ed.): Proceedings of the 1st Central-European Workshop on Services and their Composition, ZEUS 2009, Stuttgart, Germany, March 2--3, 2009.

62. Kopp, O., Martin, D., Wutke, D., Leymann, F., “On the Choice Between Graph-Based and Block-Structured Business Process Modeling Languages”, In: Modellierung betrieblicher Informationssysteme (MobIS 2008). Saarbrücken, Germany, November 27 - 28, 2008.

63. Korherr, B., “Business Process Modelling - Languages, Goals and Variabilities”, PhD Thesis, Vienna University Of Technology, 2008.

64. Kovacic, A., "Business renovation: business rules (still) the missing link," Business Process Management Journal, vol. 10, p. 158, 2004.

65. Kummer, O., Wienberg, F., “Renew - the Reference Net Workshop,” In Proceedings of the21st International Conference on Application and Theory of Petri Nets, Aarhus, Denmark, 2000, pp 87-89.

66. Kurtev, I., Bézivin, J., Aksit, M., "*Technological Spaces: an Initial Appraisal*", CoopIS, DOA'2002, Industrial track, 2002.

67. Kurtev, I., "*Adaptability of Model Transformations*", PhD Thesis, University of Twente, 2005.

68. Kühne, T., "*Matters of (Meta-)Modeling*", Journal on Software and Systems Modeling, Volume 5, Number 4, 369-385, December 2006.

69. Lee, S., Kim, Y., T., Kang, D., Kim, K., Lee, Y., J., "Composition of executable business process models by combining business rules and process flows". Expert Syst. Appl., 33(1):221–229, 2007.

70. List, B., Korherr, B., “An Evaluation of Conceptual Business Process Modelling Languages”, in Proceedings of the 21st ACM Symposium on Applied Computing (SAC'06), April 2006, Dijon, France, ACM Press, 2006.

71. Lukichev, S., Giurca, A., Wagner, G., Gašević, D., Ribarić, M., "Using UML-based Rules for Web Services Modeling," *In Proceedings of the 2nd Int’l Workshop on Service Engineering at the 23rd Int’l Conference on Data Engineering,* pp. 290-298., 2007.

72. Lukichev, S., Wagner., G., "Visual rules modeling", In Irina Virbitskaite and Andrei Voronkov, editors, Proceedings of the 6th International Conference Perspectives of Systems Informatics, volume 4378 of Lecture Notes in Computer Science, pages 467–673. Springer, 2006.

73. Lukichev, S., Wagner, G., “UML-Based Rule Modeling with Fujaba”, *Proceedings of the 4th International Fujaba Days 2006, University of Bayreuth, Germany*. Pages: 31 – 35, 2006.

74. MagicDraw UML, No Magic, Inc., 2009, [Online], Available: www.magicdraw.com.

75. Mayer, R., Menzel, C., Painter, M., Perakath, B., de Witte, P., Blinn, T., “Information Integration for Concurrent Engineering (IICE)”, IDEF3 Process Description Capture Method Report. Technical Report September, 1995, online: http://www.idef.com/pdf/idef3_fn.pdf.

76. McBrien, P., Seltveit, H., A., "Coupling Process Models and Business Rules," in Information Systems Development for Decentralized Organizations, Proceedings of the IFIP 8.1 WG Conference: Chapman Hall, 1995.

77. MDT-UML2Tools, Galileo Project, Eclipe foundation, 2009, [Online], Available: http://www.eclipse.org/modeling/mdt/?project=uml2tools#uml2tools.

78. Mendling, J., Neumann, G, Nuttgens, M., “A Comparison of XML Interchange Formats for Business Process Modelling”, In: Proceedings of EMISA 2004 - Information Systems in E-Business and E-Government. LNI. 2004.

79. Mendling, J., Nüttgens, M., “EPC Markup Language (EPML) - An XML-Based Interchange Format for Event-Driven Process Chains (EPC)”, Technical Report JM-2005-03-10. Vienna University of Economics and Business Administration, Version 2005-03-10, http://wi.wuwien.ac.at/~mendling/publications/TR05-EPML.pdf.

80. Meng, J., Su, W., Y., S., Lam, H., Helal, A., "Achieving dynamic inter-organizational workflow management by integrating business processes, events and rules," in 35th Hawaii International Conference on System Sciences, Hawaii, 2002.

81. Metzger, A., "*A Systematic Look at Model Transformations*". Showed in: Beydeda, S., Book, M., Gruhn, V., (eds.) "*Model-Driven Software Development*", Springer-Verlag, pp. 19.-33., 2005.

82. Meyer, J. et al. (1994) "The Paradoxes of Deontic Logic Revisited: A Computer Science Perspective," *Technical Report UU-CS-1994-38*, Utrecht University.

83. Microsoft Visio, Microsoft Corp., 2010, [Online], Available: http://office.microsoft.com/en-us/visio/.

84. Milanović, M., Gašević, D., Giurca, A., Wagner, G., Devedžić, V., "Sharing OCL Constraints by Using Web Rules", *Ocl4All: Modelling Systems with OCL Workshop at ACM/IEEE 10th International Conference On Model Driven Engineering Languages and Systems (MODELS 2007)*, Nashville, USA, 2007.

85. Miller, J., Mukerji, J., (eds.) "MDA Guide Version 1.0.1", OMG, 2003.

86. Nitzsche, J., Lessen, T. v., and Leymann, F., "WSDL 2.0 Message Exchange Patterns: Limitations and Opportunities", In Proceedings of the 2008 Third international Conference on internet and Web Applications and Services - Volume 00 (June 08 - 13, 2008). ICIW. IEEE Computer Society, Washington, DC, 168-173. DOI= http://dx.doi.org/10.1109/ICIW.2008.80, 2008.

87. OASIS, "UDDI Version 3.2" specification, 2004, [Online], Available: http://www.uddi.org/pubs/uddi_v3.htm.

88. OMG, "Business Process Modeling Notation" v1.2 - Final Adopted Specification, [Online], Available: http://www.omg.org/docs/formal/09-01-03.pdf, January 2009.

89. OMG, "*Business Process Model and Notation (BPMN) Specification 2.0*", OMG Document Number: dtc/2010-06-05, Beta 2, [Online], Available: http://www.omg.org/spec/BPMN/2.0/Beta2/, may 2010.

90. OMG, "*Business Process Definition MetaModel Volume I: Common Infrastructure*" & "*Business Process Definition MetaModel Volume II: Process Definitions*", OMG Documents: formal/2008-11-03 & formal/2008-11-04, 2008.

91. OMG, "*Production Rule Representation (PRR)*", Beta - OMG adopted specification, [Online], Available: http://www.omg.org/docs/dtc/07-11-04.pdf, November 2007.

92. OMG, "Meta Object Facility (MOF) Core", v2.0, OMG Document formal/06-01-01, http://www.omg.org/cgi-bin/doc?formal/2006-01-01, 2005.

93. OMG, "MOF 2.0/XMI Mapping Specification", v2.1, formal/05-09-01, OMG, 2005. [Online]. Available: http://www.omg.org/docs/formal/05-09-01.pdf

94. OMG, "Object Constraint Language", OMG Specification, Version 2.0, formal/06-05-01, http://www.omg.org/docs/formal/06-05-01.pdf, 2006.

95. OMG, "Semantics of Business Vocabulary and Business Rules Specification", OMG Adopted Specification, 2006.

96. OMG, "Unified Modeling Language 2.0", Docs. formal/05-07-04 & formal/05-07-05, 2005.

97. Orriëns, B., Yang, J., Papazoglou, P., M., "A Framework for Business Rule Driven Web Service Composition", In Conceptual Modeling for Novel Application Domains, Volume 2814/2003, pp. 52-64, Springer, 2003.

98. Oryx editor, 2009, [Online], Available: http://bpt.hpi.uni-potsdam.de/Oryx/WebHome.

99. Papazoglou, P., M., Traverso, P., Dustdar, S., Leymann, F., "Service-Oriented Computing: State of the Art and Research Challenges," Computer, vol. 40, no. 11, pp. 38-45, Oct., 2007.

100. Peterson, L., J., "Petri net theory and the modeling of systems", Prentice Hall, Englewood Cliffs, New Jersey, USA, 1981.

101. Petri Net Kernel home page, 2009 [Online]. Available: http://www2.informatik.hu-berlin.de/top/pnk/.

102. Petri, C., A., "Kommunikation mit Automaten", Dissertation, Schriften des IIM Nr. 2, Institut für Instrumentelle Mathematik, Universität Bonn, 1962.

103. Pi4Tech WS-CDL Tools Suite, 2008, [Online], Available: http://sourceforge.net/projects/pi4soa.

104. Pilone, D., Pitman, N., "*UML 2.0 in a Nutshell*", O'Reilly, 2005.

105. Prezel, V., Gašević, D., Milanović, M., "Representational Analysis of Business Process and Business Rule Languages", 1st International Workshop on Business Models, Business Rules and Ontologies (BuRO 2010), co-located with the 4th International Conference on Web Reasoning and Rule Systems, RR2010, Bressanone/Brixen, Italy, September 22-24, 2010.

106. Procap software modeling tool, KBSI Tools, 2009 [Online], Available: http://www.kbsi.com/Software/KBSI/ProCap.htm.

107. Puhlmann, F., "On the Application of a Theory for Mobile Systems to Business Process Management". Doctoral Thesis, University of Potsdam, Germany (2007).

108. Recker, J., Indulska, M., Rosemann, M., Green, P., "How Good is BPMN Really Insights from Theory and Practice", in Proceedings 14th European Conference on In-formation Systems", Goeteborg, Sweden, 2006.

109. Regev, G., Soffer, P., Schmidt, R., "Taxonomy of flexibility in business processes", Produced as a result of the BPMDS'05 workshop, 2005, [Online], Available at http://lamswww.epfl.ch/conference/bpmds06/taxbpflex.

110. REWERSE I1 Rule Markup Language (R2ML). http://oxygen.informatik.tu-cottbus.de/rewerse-i1/?q=node/6, 2008.

111. Ribarić, M., Gašević, D., Milanović, M., Guirca, A., Lukichev, S., Wagner, G., "Model-Driven Engineering of Rules for Web Services," In Lämmel, R., Saraiva, J., & Visser, J. (Eds.) Post-Proceedings of 2nd Summer School on Generative and Transformational Techniques in Software Engineering II 2007, LNCS5235, pp. 377-395, Springer, 2008.

112. Rojo, M., G., Rolón, E., Calahorra, L., García, F., O., Sánchez, R., P., Ruiz, F., Ballester, N., Armenteros, M., Rodríguez, T., Espartero, R.,M., "Implementation of the Business Process Modelling Notation (BPMN) in the modelling of anatomic pathology processes", *Diagn. Pathol.*, 2(1s), s22., 2008.

113. Rosenberg, F., Dustdar, S., "Business Rules Integration in BPEL - A Service-Oriented Approach", In: Proceedings of the 7th International IEEE Conference on E-Commerce Technology (CEC'05), 19. - 22. July 2005, Munich, Germany.

114. Rosenberg, F., Dustdar, S., " Design and Implementation of a Service-Oriented Business Rules Broker", In *Proceedings of the Seventh IEEE international Conference on E-Commerce Technology Workshops* (July 19 - 19, 2005). CECW. IEEE Computer Society, Washington, DC, 55-63.

115. Rosenberg, F., Dustdar, S., "Towards a distributed service-oriented business rules system", In ECOWS '05: Proceedings of the Third European Conference on Web Services, page 14, Washington, DC, USA, 2005. IEEE Computer Society.

116. Russell, N., ter Hofstede, A.H.M., van der Aalst, W.M.P., Mulyar, N., "Workflow Control-Flow Patterns: A Revised View", BPM Center Report BPM-06-22 , BPMcenter.org, 2006.

117. Scheer, W., A., "ARIS Business Process Modeling", Springer-Verlag, 1999.

118. Schmidt, R., "Web services based execution of business rules", In Proc. of the Intl. Workshop on Rule Markup Languages for Business Rules on the Semantic Web - RuleML, 2002.

119. Schnieders, A., Puhlmann, F., "Variability mechanisms in e-business process families", in *Abramowicz & Mayr (eds.), Proceedings of the 9th International Conference on Business Information Systems (BIS 2006),* volume P-85 of LNI, Bonn, Gesellshaft für Informatik, 2006, pp. 583-601.

120. Seel, C., Vanderhaeghen, D., "Meta-Model based Extensions of the EPC for Inter-Organisational Process Modelling", 4. GI-Workshop "EPK 2005 – Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten", Hamburg 2005.

121. Seidewitz, E., "What Models Mean", IEEE Software, pp. 26-32, 2003.

122. Selic, B., "*The Pragmatics of Model-Driven Development*", IEEE Software (spec. issue on Model Driven Development), Vol. 20, No. 5, 2003, pp. 19.-25.

123. Starfield, M., Smith, K., A., Bleloch, A., L., "*How to model it: Problem Solving for the Computer Age*", McGraw-Hill, New York, 1990.

124. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E., "EMF: Eclipse Modeling Framework", Addison-Wesley Professional, 2nd Edition, 2008.

125. Steinke, G., Nikolette, C., "Business rules as the basis of an organization's information syste*m", Industrial management + Data Systems,* vol. 103, p. 52, 2003.

126. Strahonja, V., "The Evaluation Criteria of Workflow Metamodels", In Proceedings of the ITI 2007 29th Int. Conf. on Information Technology Interfaces, June 25-28, 2007, Cavtat, Croatia.

127. "*Structured Query Language (SQL) standard*", InterNational Commitee for Information Technology Standards (INCITS), American National Standard ANSI/ISO/IEC 9075-2:1999, September 1999.

128. Taveter, K., "A Multi-Perspective Methodology for Agent-Oriented Business Modelling and Simulation", Doctoral thesis, Tallinn University of Technology, 2004.

129. Taveter, K., Wagner, G., “Towards Radical Agent-Oriented Software Engineering Processes Based on AOR Modelling”, In: Henderson-Sellers, B., Giorgini, P. (eds.), *Agent-Oriented Methodologies*. Idea Group Publishing, 2005.

130. The Business Rules Group, Business Rules Manifesto - The Principles of Rule Independence ver. 1.2 (Jan. 8, 2003). [Online], Available: www.BusinessRulesGroup.org.

131. Timm, J., Gannod, G., “A Model-Driven Approach for Specifying Semantic Web Services”, In Proc. of IEEE International Conference on Web Services (pp. 313–320), 2005.

132. "*UML Profile and Interchange Models for Enterprise Application Integration (EAI) Specification*", OMG Formal Specification, March 2004, formal/04-03-26.

133. "*UML Profile for Metaobject Facility (MOF) Specification*", OMG adopted specification, February 2004, Version 1.0, formal/04-02-06. [Online], Available: http://www.omg.org/docs/formal/04-02-06.pdf.

134. W3C, SOAP Ver. 1.2 Part 1: Messaging Framework. W3C Recommendation, [Online], Available: http://www.w3.org/TR/soap12-part1/.

135. W3C, Web Services Glossary, Working Group Note 11 February 2004, Online, Available: http://www.w3.org/TR/ws-gloss/.

136. W3C, Web Services Description Language (WSDL) Ver. 2.0 Part 1: Core Language. W3C Candidate, [Online], Available: http://www.w3.org/TR/2007/REC-wsdl20-20070626.

137. W3C, Web Services Description Language (WSDL) Version 2.0 Part 2: Adjuncts, W3C Recommendation 26 June 2007, [Online], Available: http://www.w3.org/TR/wsdl20-adjuncts.

138. W3C, Web Services Description Language (WSDL) Version 2.0: Additional MEPs, W3C Working Group Note 26 June 2007, [Online], Available: http://www.w3.org/TR/wsdl20-additional-meps/.

139. W3C, Web Services Policy Framework 1.2 (WS-Policy), W3C Member Submission 25 April 2006, [Online], Available: http://www.w3.org/Submission/WS-Policy/.

140. “What is a business rule ?”, Business Rules Group, 2008, [Online], Available: http://www.businessrulesgroup.org/defnbrg.shtml.

141. Wagner, G., Tabet, S., Boley, H., "*MOF-RuleML: The Abstract Syntax of RuleML as a MOF model*", Integrate 2003, OMG Meeting, Boston, October 2003.

142. Wagner, G., Giurca, A., Lukichev, S., “A Usable Interchange Format for Rich Syntax Rules Integrating OCL, RuleML and SWRL”, *In Proceedings of WWW2006 conference*, Edinburgh, UK, 2006.

143. Wagner, G., Giurca, A., Lukichev, S., "*A General Markup Framework for Integrity and Derivation Rules*", Dagstuhl Seminar Proceedings, Principles and Practices of Semantic Web Reasoning, 2006. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2006/479.

144. Wand Y., Weber, R., “On the ontological expressiveness of information systems analysis and design grammars, Science And Technology”, vol. 3, pp. 217-237, 1993.

145. Weerawarana, S., et al., “Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More”, Prentice Hall, 2005.

146. Weske, M., Business Process Management, Springer, 2007.

147. White, A., S., “Mapping BPMN to BPEL Example”, IBM, online: http://www.bpmn.org/Documents/Mapping%20BPMN%20to%20BPEL%20Example.pdf, February 2005.

148. Wohed, P., van der Aalst, M., P., W., Dumas, M., ter Hofstede, M., H., A., Russell, N., “On the Suitability of BPMN for Business Process Modelling”, In: 4th International Conference on Business Process Management, 5-7 September 2006, Vienna, Austria.

149. Workflow Process Definition Interface - XML Process Definition Language (XPDL), Version 2.1, 2009, Online: http://www.wfmc.org/xpdl.htm.

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

151. zur Muehlen, M. (2004). Workflow-based Process Controlling. Berlin, Logos Verlag.

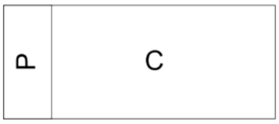
152. zur Muehlen, M., Indulska, M., Kamp, G., "Business Process and Business Rule Modeling: A Representational Analysis",: In Taveter, K.; Gasevic, D. (Eds.): The 3rd International Workshop on Vocabularies, Ontologies and Rules for The Enterprise (VORTE 2007). Baltimore, MD, October 15th, 2007.

Appendix A. Basic mappings between BPMN and WS-BPEL

Not all BPMN process models could be mapped into BPEL orchestrations, as BPEL cannot support some constructs, such as unstructured loops [89]. In addition, an important issue is that a BPMN model can be mapped into BPEL, only if it does not contain any deadlocks (point in a process that contain tokens that cannot be removed), or a lack of synchronization (point in a process with more then one token). In the following section, for representation of the BPMN and BPEL mappings, we use structure from [89].

Process and Activities

The first BPMN element that should be mapped to BPEL is pool. A BPMN pool is mapped to a BPEL <*process*> element. The name of the pool is the same as the name of the process, while the pool contents are placed into the <*process*> element (usually after *partnerLink*, *variable* and *correlationSets* declarations). This mapping is shown in Figure 225.

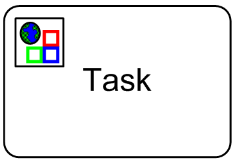


```
<process name="[P-name]"
     targetNamespace="[targetNamespace]"
     expressionLanguage="[expressionLanguage]"
     suppressJoinFailure="yes"
     xmlns="http://docs.oasis-
            open.org/wsbpel/2.0/process/executable">
   [C]
</process>
```

a) BPMN pool b) Corresponding BPEL code

Figure 225. Mapping of the BPMN pool into corresponding BPEL code

A BPMN *Service* task is mapped into a BPEL <*receive*> activity (as shown in Figure 226). The <*receive*> activity is used in BPEL to wait for a message to arrive.

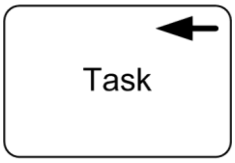


```
<receive name=”[Task]”
          createInstance=”[instantiate? ‘yes’”’no’”
          partnerLink=”...”
          portType=”...”
          operation=”...”>
</receive>
```

a) BPMN Service task b) Corresponding BPEL code

Figure 226. Mapping of the BPMN Service task into corresponding BPEL code

A BPMN *Receive* task is also mapped into a BPEL <*receive*> activity (see Figure 227).

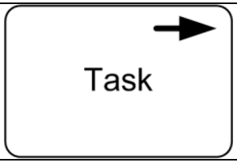


```
<receive name=”[Task]”
          createInstance=”[instantiate? ‘yes’”’no’”
          partnerLink=”[Task-service-ref]”
          portType=”[Task-operation-interface]”
          operation=”[Task-operation]”>
</receive>
```

a) BPMN Receive task b) Corresponding BPEL code

Figure 227. Mapping of the BPMN Receive task into corresponding BPEL code

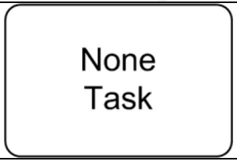
A BPMN *Send* task is mapped into a BPEL <*invoke*> activity (see Figure 228). The <*invoke*> activity in BPEL is used to invoke an operation on a *portType*, offered by a participant.

	<**receive** name="[Task]"
	        partnerLink="..."
	        portType="..."
	        operation="...">
	</**receive**>

a) BPMN Send task	b) Corresponding BPEL code

Figure 228. Mapping of the BPMN Send task into corresponding BPEL code

A BPMN *None* task is mapped into a BPEL <*empty*> activity (see Figure 229). The <*empty*> activity in BPEL is “no-op” in a process.

	<**receive** name="[Task]"
	        partnerLink="..."
	        portType="..."
	        operation="...">
	</**receive**>

a) BPMN None task	b) Corresponding BPEL code

Figure 229. Mapping of the BPMN None task into corresponding BPEL code

A BPMN *Message* structure is mapped is mapped into the BPEL, by using WSDL <*message*> type (see Figure 230).

<**Message** name="name">
  <StructureDefinition typeLanguage =
      http://wwww3.org/2001/XMLSchema>
  </StructureDefinition>
</**Message**>
	<wsdl:**message** name = "[name]">
	  [xmlSchema]
	</wsdl:**message**>

a) BPMN Message structure	b) Corresponding WSDL message type

Figure 230. Mapping of the BPMN Message type into corresponding WSDL message type

A BPMN *interface*, with its operation, is mapped to a corresponding WSDL *portType* with corresponding operations (see Figure 231).

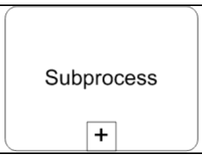
<**Interface** name="name">
  <Operations>
    <Operation name="opname1">
      <inMessageRef ref="msg1nameI"/>
       <outMessageRef ref="msg1nameO"/>
       <errorRef ref="errorName1"/>
       ...
     </Operation>
     ...
  </Operations>
</**Interface**>
	<wsdl:**portType** name = "[name]">
	 <operation name="[opname1]">
	  <wsdl:input message="msg1nameI"/>
	  <wsdl:output message="msg1nameO"/>
	  <wsdl:fault name="errorName1"/>
	  ...
	 </operation>
	  ...
	</wsdl:**portType**>

a) BPMN Interface	b) Corresponding WSDL portType

Figure 231. Mapping of the BPMN Interface into corresponding WSDL portType

For sending or receiving BPMN messages that have an associated correlation set, a BPMN key-based correlation set (*KeyBasedCorellationSet*) is mapped to a corresponding BPEL *correlationSet*. A BPEL property alias *messageType* is attained from the message structure definition of the BPMN message key expression, and the message part name is attained from the message key expression [89]. The BPEL *correlation* element is set based on the associated key-baed correlation set of the BPMN activity, depending whether a message flow initiates or participates in an associated conversation.

A BPMN *subprocess* is mapped into a BPEL <*scope*> activity (see Figure 232). The BPEL <*scope*> activity is used to define a nested activity with its own associated <*partnerLinks*>, <*messageExchanges*>, <*correlationSets*>, <*faultHandlers*>, <*variables*>, <*terminationHandler*>, <*compensationHandler*>, and <*eventHandlers*>.

	`<scope>`
	`  [Subprocess]`
	`</scope>`

a) BPMN Subprocess	b) Corresponding BPEL code

Figure 232. Mapping of the BPMN Subprocess into corresponding BPEL scope

While the BPMN subprocess that containts an event for its first element, is mapped into a BPEL *<scope>* with a corresponding handler as follows: a BPMN message event is mapped to a BPEL *onEvent <eventHandlers>*, a BPMN timer event is mapped to a BPEL *onAlarm <eventHandlers>*, a BPMN error event is mapped to a BPEL *<faultHandlers>*, and a BPMN compensation event is mapped to a BPEL *<compensationHandler>*.

BPMN standard loops (repeating activity) with a *testTime* attribute "Before" or "After" execution of an activity are mapped to BPEL *while* and *repeatUntil* activities, respectively. Multi-instance activities are mapped to BPEL *forEach* activities [89]. For the multi-instance activities, the *<startCounterValue>* attribute is set to 1, and the *<finalCounterValue>* is set to the value of the activities *condition* expression. The *<scope>* of a *forEach* activity is actually translated BPMN activity.

Events

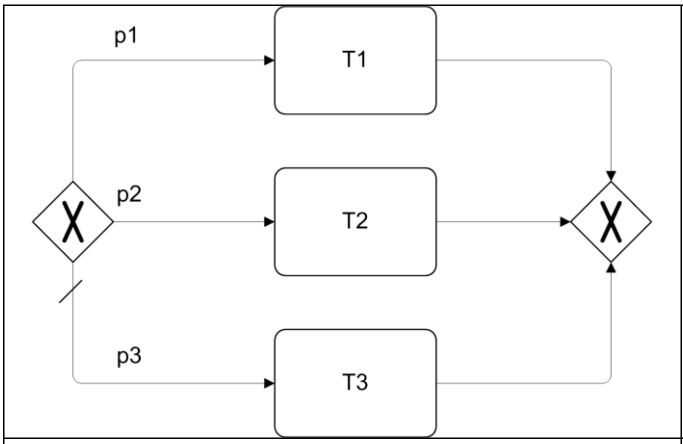
A BPMN start message event is mapped to a BPEL *<receive>* activity with the *createInstance* attribute set to "*yes*" (we want to create a process instance), while compensation and error start events can only occur in a sub-process, and their mapping is described in the previous section.

The BPMN non-boundary intermediate message event is also mapped to the BPEL *<receive>* activity, but with the *createInstance* attribute set to "*no*", as we do not want to create a process instance. The BPMN timer intermediate event is mapped to the BPEL *<wait>* activity, while the BPMN compensation intermediate event is mapped to the BPEL *<compensate>* activity if the compensation event does not reference an activity, or to the BPEL *<compensateScope>*, otherwise. The BPEL *<wait>* activity is used to wait for a certain period, until a certain point in time has been reached.

A BPMN None end event, which is used to mark end of the process, is mapped to an *<empty>* BPEL activity. A BPMN Message end event is mapped to a BPEL *<invoke>* activity, while a BPMN Error end event is mapped to a BPEL *<throw>* activity, which is used to generate a fault from the process [49]. A BPMN Compensation end event is mapped to *<compensate>* or *<compensateScope>* activities, similary to the corresponding intermediate event. A BPMN Terminate end event is mapped to a BPEL *<exit>* activity, which is used to end a business process instance [49]. There are also a few complex mappings of BPMN boundary events to the corresponding BPEL code [89].

Gateways

The BPMN Exclusive data-based gateway is mapped to the BPEL *<if>* activity, as shown in Figure 233. An *<if>* activity in BPEL is used to select exactly one activity from a set of choices based on a predefined condition.

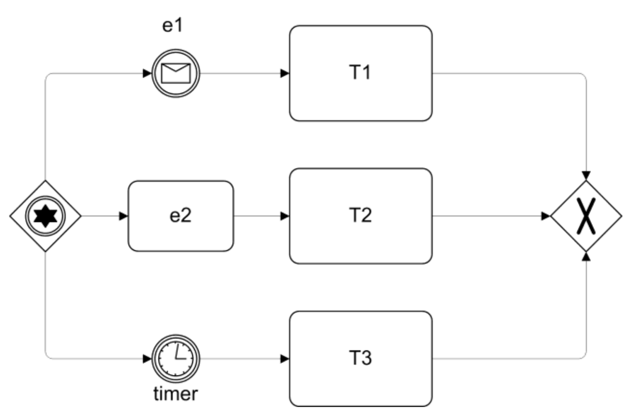


```
<if><condition>p1</condition>
    [T1]
  <elseif><condition>p1</condition>
    [T2]
  </elsif>
  <else>
    [T3]
  </else>
</if>
```

a) BPMN Exclusive data-based gateway b) Corresponding BPEL code

Figure 233. Mapping of the BPMN Exclusive data-based gateway into corresponding BPEL code

A BPMN Exclusive event-based gateway is mapped to a BPEL <*pick*> activity (see Figure 234). The <*pick*> activity is used to wait for either more messages to arrive or a time-out to occur. When either of the triggers occurs, the corresponding child activity is performed. This mapping holds for cases with more than three branches.



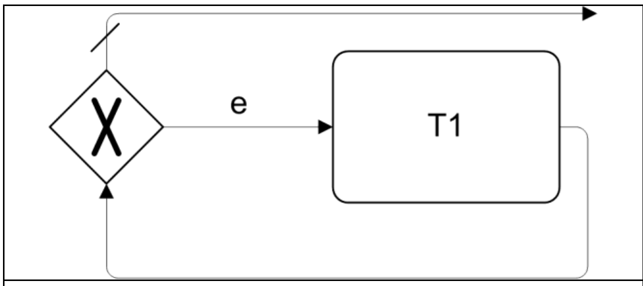
```
<pick createInstance="[Instantiate]">
  <onMessage partnerLink="e1-
participant]" operation="e1op">
    [T1]
  </onMessage>
  <onMessage partnerLink="e2-
participant]" operation="e2op">
    [T2]
  </onMessage>
  <onAlarm>
    [timer]
    [T3]
  </onAlarm>
</if>
```

a) BPMN Exclusive event-based gateway b) Corresponding BPEL code

Figure 234. Mapping of the BPMN Exclusive event-based gateway into corresponding BPEL code

A Parallel fork-join pattern with AND gateways is mapped directly to a BPEL <*flow*> activity, while the Sequence pattern is mapped directly to the BPEL <*sequence*> activity [89]. The <*flow*> activity is used to specify activities to be performed concurrently, while the <*sequence*> activity is used to define activites to be performed sequentially in lexical order.

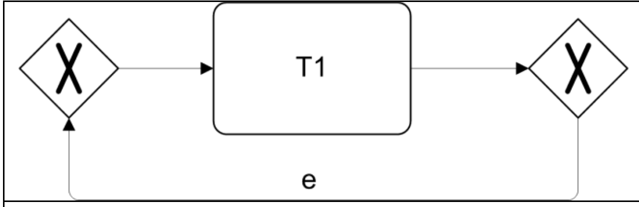
Two types of loops are supported in BPEL, namely, while and repeat loops. A while loop structure in BPMN is mapped to a <*while*> BPEL activity (see Figure 235). The <*while*> activity is used in BPEL to define an activity that should be repeated while the defined <*condition*> evaluates to true.


 	
	<**while**>
	  <condition>
	    [e]
	  </condition>
	    [T1]
	</**while**>

a) BPMN while loop structure	b) Corresponding BPEL code

Figure 235. Mapping of the BPMN while loop structure into corresponding BPEL code

A BPMN repeat loop structure is mapped to a <*repeatUntil*> BPEL activity (see Figure 236). The <*repeatUntil*> activity is similar to the <*while*> activity, but in this case, the <*condition*> is tested after the execution of the inner activity.


 	
	<**repeatUntil**>
	    [T1]
	  <condition>
	    [not e]
	  </condition>
	</**while**>

a) BPMN repeat loop structure	b) Corresponding BPEL code

Figure 236. Mapping of the BPMN repeat loop structure into corresponding BPEL code

Data

The BPMN data objects are mapped directly to the BPEL <*variable*> construct, which is defined in <*variables*> section. Also, BPMN properties are mapped to BPEL variables [89]. The input set is mapped to a WSDL message defining the input of a BPEL activity, for a Send and Service tasks, while for a Receive task a single output is mapped to a WSDL message defining the output of the BPEL activity (maps message parts of WSDL message). These mappings are shown in Figure 237.

<**inputSet** name=”iset”>
  <dataInput name=”input”>
    <structureDefinition
structure=”type”/>
  </dataInput>
  ...
</**inputSet**>
	<wsdl:**message** name=”iset-name”>
	  <part name=”[input-name]”
	type=”[type]”/>
	  ...
	</wsdl:**message**>

<**outputSet** name=”oset”>
  <dataOutput name=”output”>
    <structureDefinition
structure=”type”/>
  </dataOutput>
  ...
</**outputSet**>
	<wsdl:**message** name=”oset-name”>
	  <part name=”[output-name]”
	type=”[type]”/>
	  ...
	</wsdl:**message**>

a) BPMN input and output sets	b) Corresponding BPEL code

Figure 237. Mappings of the BPMN input and output sets into corresponding BPEL code

Incoming Data associations for a Service task are mapped to a corresponding <*toParts*> of the <*invoke*> activity, while the outgoing Data associations are mapped to the corresponding <*fromParts*> of the same <*invoke*> activity. The <*toParts*> element is used to create a multi-part WSDL mesasge from BPEL variables (i.e., anonymous WSDL variable). Incoming Data associations for a Send task are also mapped to the <*toParts*>. However, outgoing Data associations for a Receive task, are mapped to <*fromParts*> of the <*receive*> activity. The <*fromParts*> element is similar to the <*toParts*> element, but in this case <*fromPart*> element is used to retrieve data from an incoming WSDL message and to place it in to a BPEL variable.

Appendix B. Mapping rBPMN constructs to choreographies (BPEL4Chor)

This mapping is organized in three main parts: generation of participant types in a participant topology; creation of participant references and participant sets; and generation of message links from the message flow.

Participant types

Each rBPMN pool is directly mapped to a participant type. In Figure 238, we show how a rBPMN participant (pool) is mapped to a corresponding BPEL4Chor *participantType* with the same name in the *participantTypes* section in a BPEL4Chor *topology*.

```
<topology name="topo"
     targetNamespace="[targetNamespace]" ...>
   <participantTypes>
      <participantType name = "Pool" .../>
      ...
   </participantTypes>
   ...
</topology>
```

a) rBPMN pool

b) Corresponding BPEL4Chor code

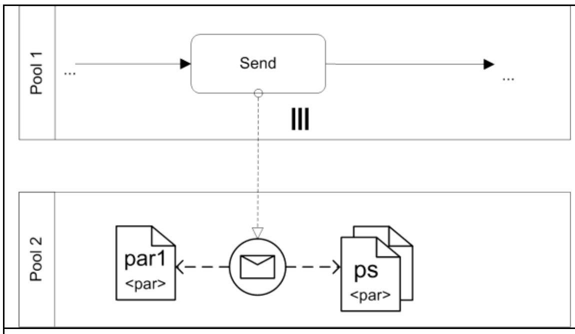
Figure 238. Paticipant type in BPEL4Chor

Participant references and participant sets

Participant sets and references are used to represent instances of Participant types. As we described in section 3, we integrated these two concepts into one (Participant set). Therefore, we use Participant sets to represent one or more references for a given participant type, and in the case if the number of participants is not defined at design-time [24].

If a pool is defined as a multi-instance pool, then such a pool is defined by a *participantType* and is referenced by a *participantSet*. However, if a pool is not defined as a multi-instance pool, then it is defined by a *participantType*, referenced by a *Participant*. Participant references and Participant sets can be associated with BPEL activities to define a participant role in a communication.

In Figure 239, we have an association from a receiving message flow from Pool 1 to Participant set *ps*. The actual participant reference in the set is represented by the *ps* participant Set data object. The *par1* participant reference contained in the set is one particular participant from the set of participants.



```
<participantSet>
   name="ps" type="Pool 1">
    <participant name="par1"/>
</participantSet>
...
```

a) Send message in rBPMN

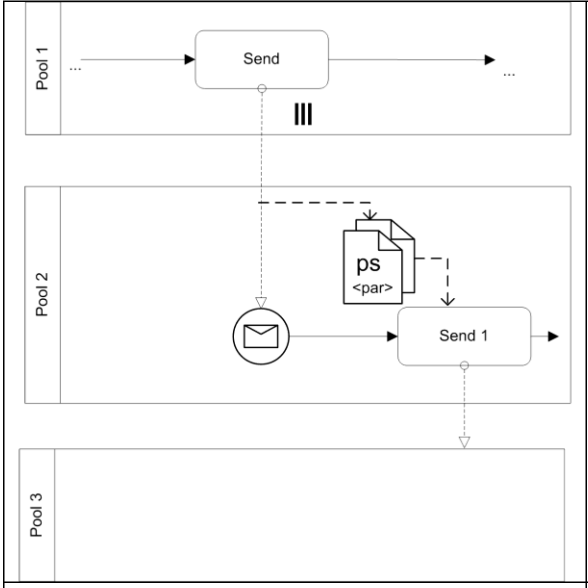
b) Corresponding BPEL4Chor code

Figure 239. Storing participant reference in a Participant set

Generation of message links

Message links are used to define which participants can communicate with other particular participants [24], where message links connect participants given in a topology.

In Figure 240, Participant set *ps* is associated with a message flow, and the *ps* represents references which are passed over the message flow from one participant to another. The first *messageLink*, in Figure 240, contains the *participantRefs* attribute, and this attribute realize *link passing mobility*. If a message is sent from a multi-instance pool (participant), then the *senders* attribute is used in the description of a *messageLink*. In addition, if a *bindSenderTo* attribute is used in a *messageLink*, then this implies that the sender of the message must include a reference to herself in the message [24].



```
<messageLink>
   sender=”Pool 1”
   receiver=”Pool 2”
   participantRefs=”ps”
   sendActivity=”Send”/>

<messageLink>
   sender=”Pool 2”
   receiver=”Pool 3”
   sendActivity=”Send 1”.../>
...
```

a) Send message with participant set in rBPMN

b) Corresponding BPEL4Chor code

Figure 240. Link passing mobility and message links

In BPEL4Chor, the usage of *partnerLink*, *portType*, and *operation* attributes in communicating activities from BPEL are forbidden in BPEL4Chor, in order to abstract from WSDL artifacts. The communicating activities are defined by using the message links in a topology (as shown in the paragraph above).

The *participant grounding* is mapping to the web-service specific configuration, in order to use the BPEL4Chor choreography on a specific platform. A message links are grounded to WSDL operations, and participant references are grounded to WSDL properties. The WSDL property defines where an element is located in a certain message type. By using these WSDL properties, BPEL process can extract the concrete service reference from the incoming message [24].

A complete description of BPEL4Chor and its relation to BPEL can be found in [24][23].

Appendix C. Use case models in the rBPMN editor

In this section we will show use case process models from section 5 done in the rBPMN editor.

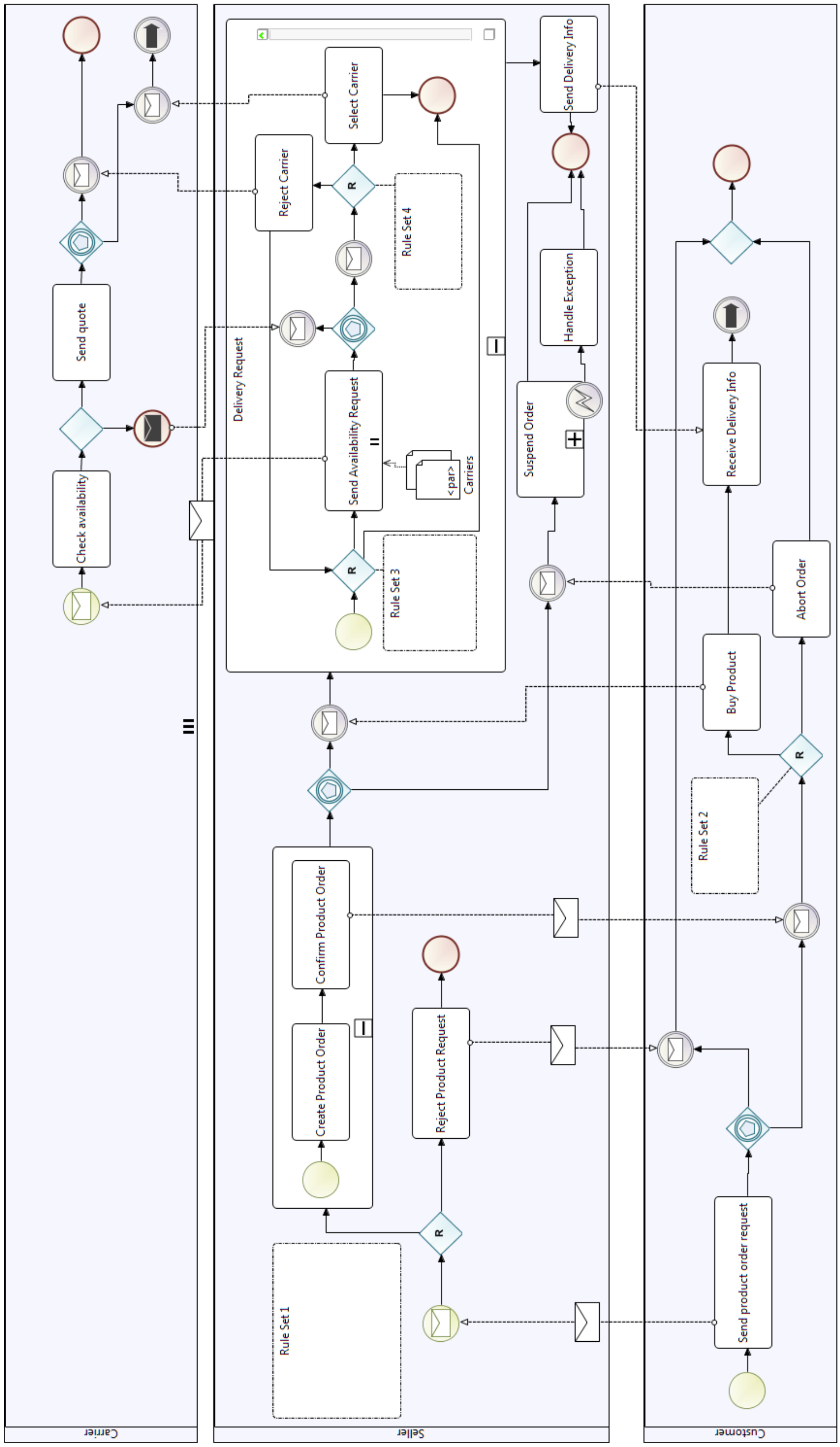


Figure 241. The on-line product order process in the rBPMN editor (from Figure 200)

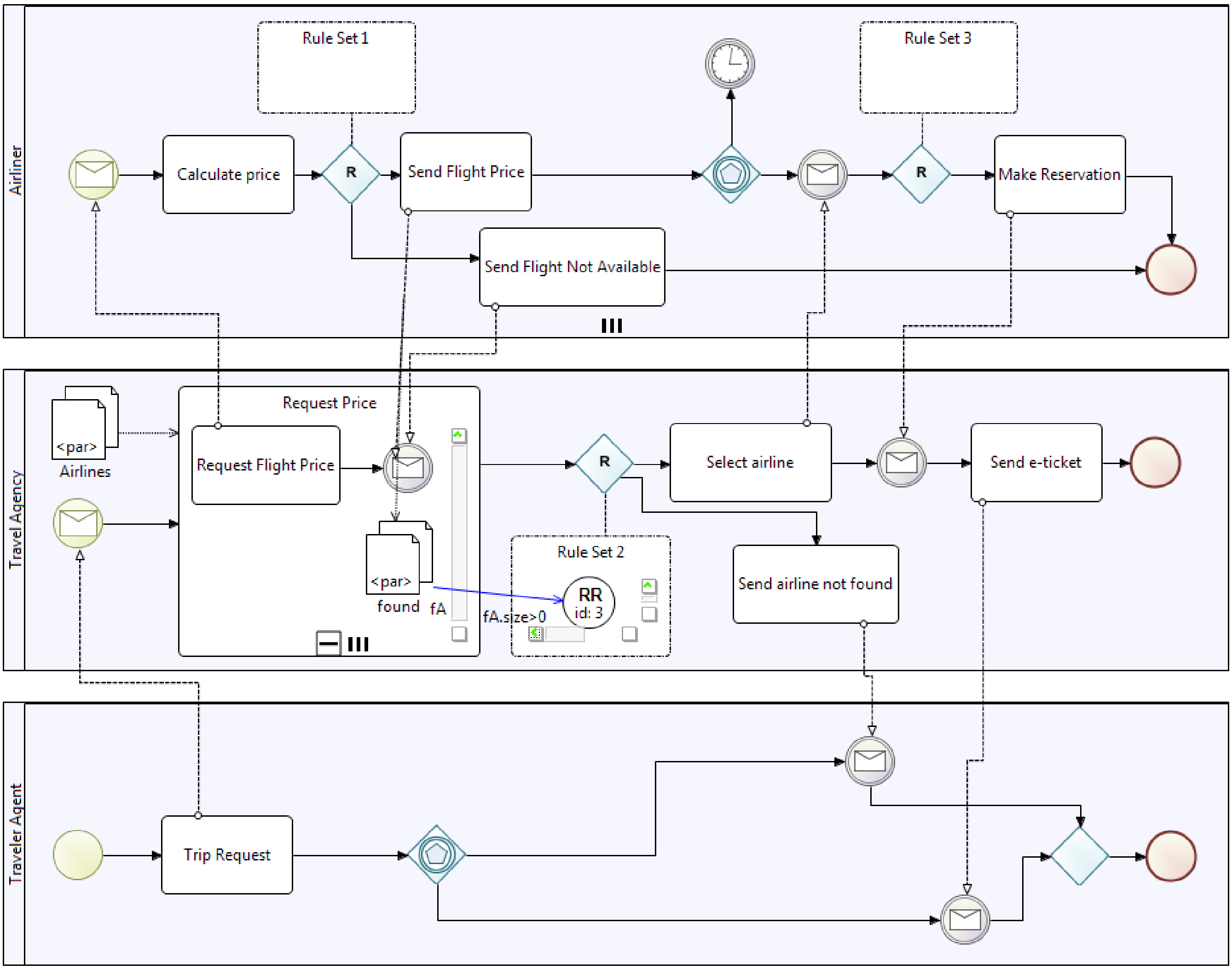


Figure 242. Interconnected behavioral choreography diagram for the Flight request process in the rBPMN editor (from Figure 205)

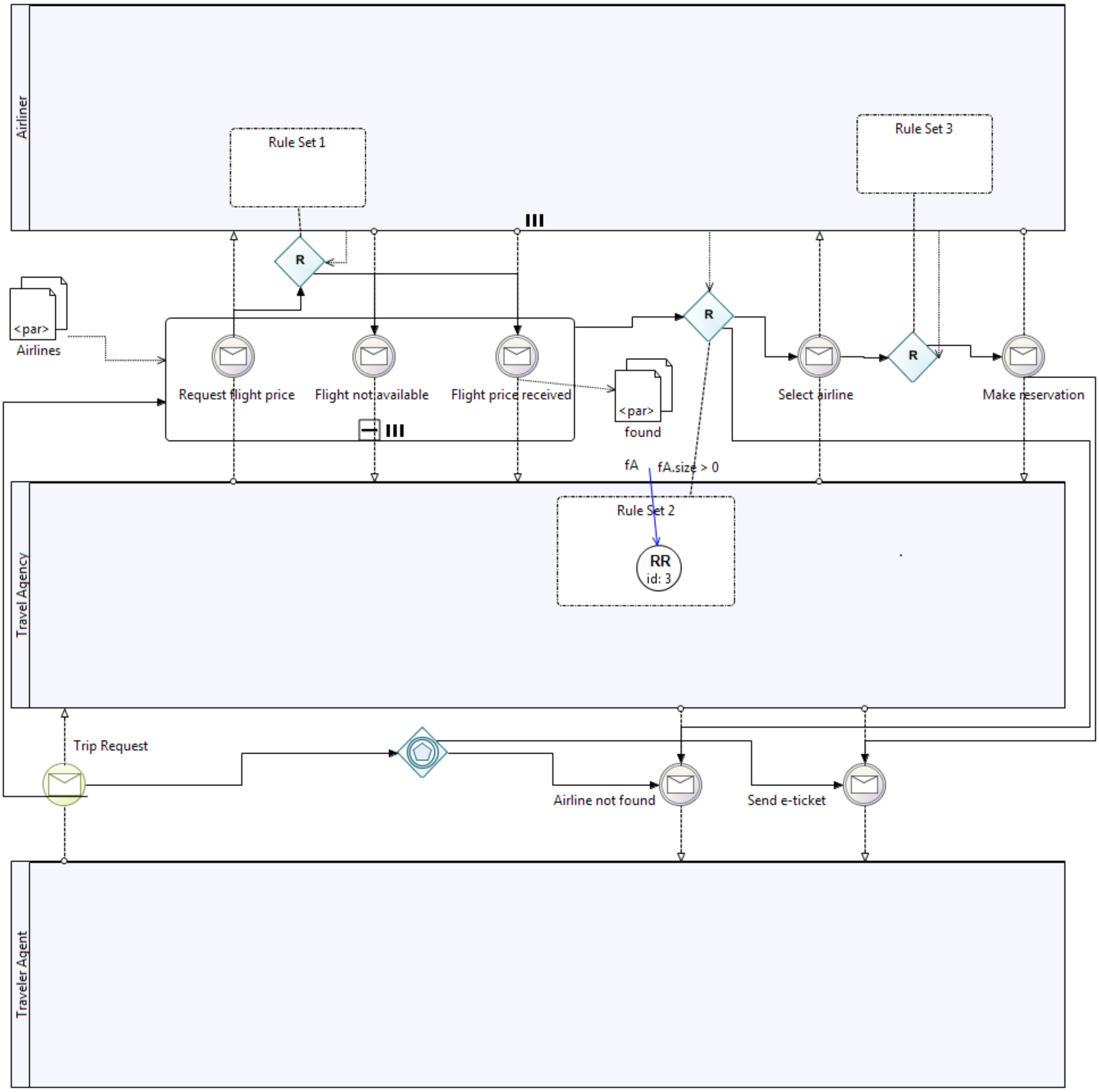


Figure 243. rBPMN interaction choreography model for the flight request process in the rBPMN editor (from Figure 209)

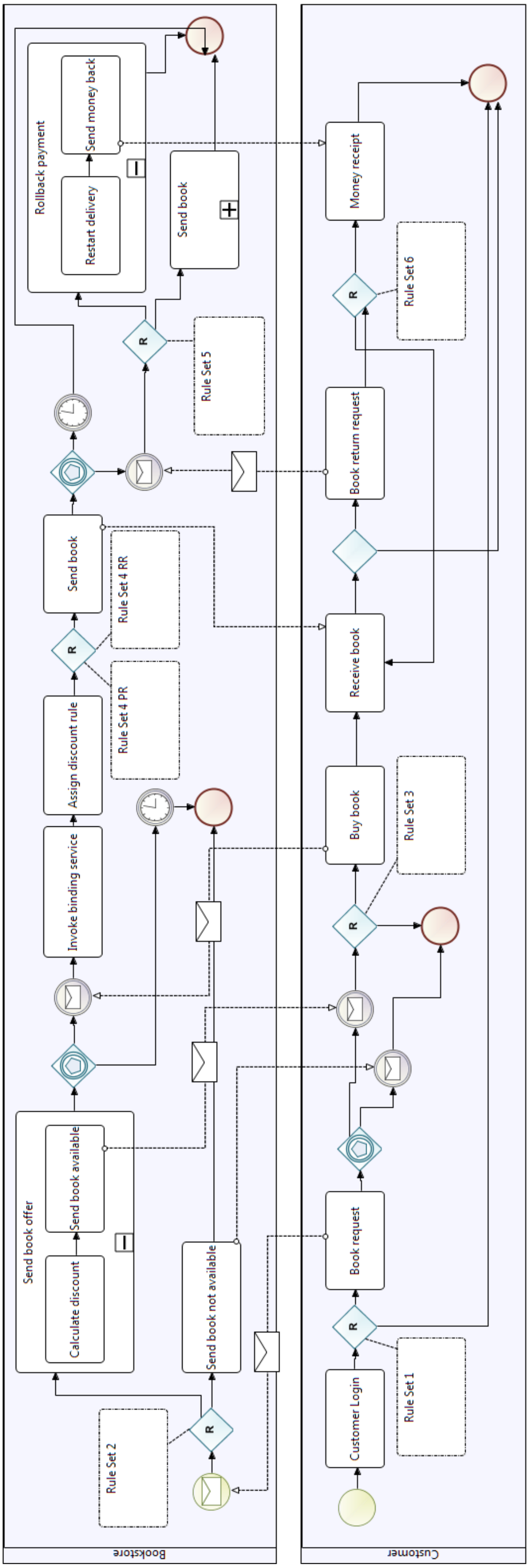
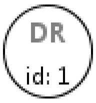
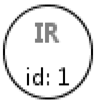
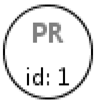
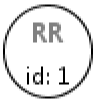
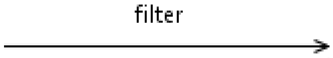
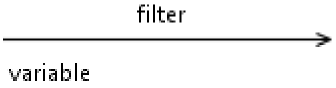
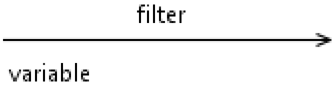
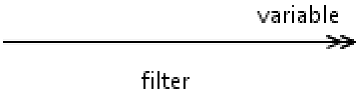
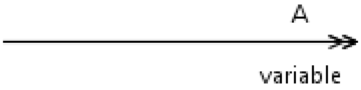
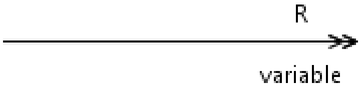
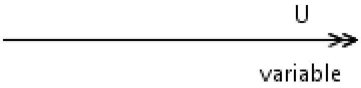
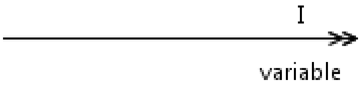
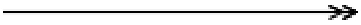
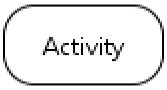
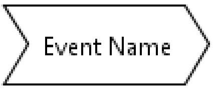
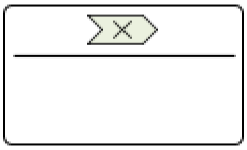
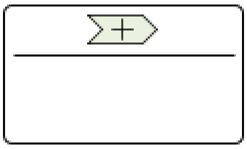
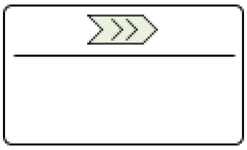
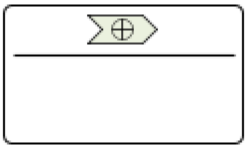
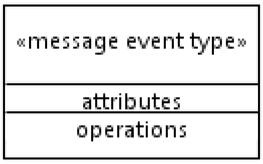
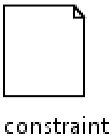
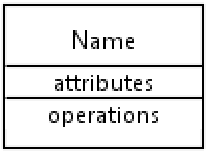
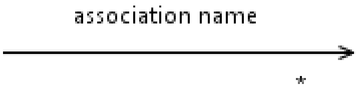
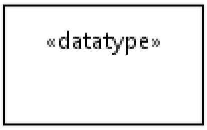
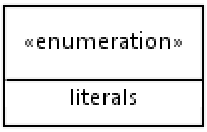


Figure 244. The book buy request scenario in the rBPMN editor (from Figure 210)

Appendix D. rBPMN graphical concrete syntax for rules

Graphical representation	Metamodel element	Constraints
	`DerivationRule`	
	`IntegrityRule`	
	`ProductionRule`	
	`ReactionRule`	
	`ReferencePropertyAtom`	Association Condition: from *ReferenceProperty* to *Rule*. Association Conclusion: from *Rule* to *ReferenceProperty*.
	`PropertyAtom`	Property Condition: from *Property* to *Rule* Property Conclusion: from *Rule* to *Property*
	`ObjectClassificationAtom`	Classification Condition: from *Class* to *Rule* Classification Conclusion: from *Rule* to *Class*
	`ObjectClassificationAtom`	Post Condition: from a *ReactionRule* or *ProductionRule*, to a *Class.*
	`AssertActionExpression`	Assert Action: from a *ReactionRule* or *ProductionRule*, to a *Class.*
	`RetractActionExpression`	Retract Action: from a *ReactionRule* or *ProductionRule*, to a *Class.*
	`UpdateActionExpression`	Update Action: from a *ReactionRule* or *ProductionRule*, to a *Class.*
	`InvokeActionExpression`	Invoke Action: from a *ReactionRule* or *ProductionRule*, to a *Class.*
	`InvokeActionExpression`	Invoke Activity Action: from a *ReactionRule* or *ProductionRule*, to a *ActionEventExpression.*
	`ActionEventExpression`	

	`AtomicEventExpression`	
	`triggeringEventExpr`	
	`triggeredEventExpr`	
	`ChoiceEventExpression`	
	`ParallelEventExpression`	
	`SequenceEventExpression`	
	`AndNotEventExpression`	
	`MessageType`	
	`OCLInvariant`	
	`Class`	
	`ReferenceProperty`	Class Association: from a *Class* to a *Type*.
	`Datatype`	

	EnumerationDatatype	