UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
DEPARTAMENTO DE INFORMÁTICA E MATEMÁTICA APLICADA
DOUTORADO EM CIÊNCIA DA COMPUTAÇÃO

**Plácido Antônio de Souza Neto**

# *A Methodology for Building Reliable Service-Based Applications*

Natal-RN / Brasil

2012

**Plácido Antônio de Souza Neto**

# *A Methodology for Building Reliable Service-Based Applications*

Tese submetida ao Programa de Pós-Graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte (UFRN), como requisito para a obtenção do grau de Doutor em Ciência da Computação.

Advisor: Prof. Dr. Martin Alejandro Musicante (UFRN-Brazil)

Co-Advisor: Prof. Dra. Genoveva Vargas-Solar (CNRS-France)

Natal-RN / Brasil

2012

*I dedicate this work to my beloved wife Ana Flávia and to my son Lucas Kauai, my daily sunshine.*

# *Resumo*

Esta tese apresenta $\pi$SOD-M (Policy-based Service Oriented Development Methodology), uma metodologia para a modelagem de aplicações orientadas a serviços associandas a *Políticas* de qualidade. O trabalho propõe um método orientado a modelo para desenvolvimento de aplicações confiáveis. $\pi$SOD-M consiste de: (i) um conjunto de meta-modelos para representação de requisitos não-funcionais associados a serviços nos diferentes níveis de modelagem, a partir de um modelo de caso de uso até um modelo de composição de serviço, (ii) um meta-modelo de plataforma específica que representa a especificação das composiçoes e as políticas, (iii) regras de transformação model-to-model e model-to-text para semi-automatizar a implementação de composiçoes de serviços confiáveis, e (iv) um ambiente que implementa estes meta-modelos e regras, representando assim aspectos transversais e limitações associadas a serviços, que devem ser respeitados. Esta tese também apresenta uma classificação e nomenclatura de requisitos não-funcionais para o desenvolvimento de aplicações orientadas a serviços. Nossa abordagem visa agregar valor ao desenvolvimento de aplicações orientadas a serviços que têm necessidades de garantias de requisitos de qualidade. Este trabalho utiliza conceitos das áreas de desenvolvimento orientado a serviços, design de requisitos não-funcionais e desenvolvimento dirigido a modelos para propor uma solução que minimiza o problema de modelagem de serviços web confiáveis.

**Área de Concentração**: Engenharia de Software
**Palavras-chave**: serviços confiáveis, políticas, desenvolvimento dirigido a modelos, metodologia, $\pi$SOD-M.

# *Abstract*

This thesis presents $\pi$*SOD-M* (*Policy-based Service Oriented Development Methodology*), a methodology for modeling reliable service-based applications using *policies*. It proposes a model driven method with: (i) a set of meta-models for representing non-functional constraints associated to service-based applications, starting from an use case model until a service composition model; (ii) a platform providing guidelines for expressing the composition and the policies; (iii) model-to-model and model-to-text transformation rules for semi-automatizing the implementation of reliable service-based applications; and (iv) an environment that implements these meta-models and rules, and enables the application of $\pi$SOD-M. This thesis also presents a classification and nomenclature for non-functional requirements for developing service-oriented applications. Our approach is intended to add value to the development of service-oriented applications that have quality requirements needs. This work uses concepts from the service-oriented development, non-functional requirements design and model-driven delevopment areas to propose a solution that minimizes the problem of reliable service modeling. Some examples are developed as proof of concepts.

**Area of Concentration**: Software Engineering

**Key words**: reliable service, policy, model-driven development, methodology, $\pi$SOD-M.

# *Contents*

# *List of Figures*

# *List of Tables*

# 1 Introduction

"*The greatest enemy of knowledge is not ignorance, it is the illusion of knowledge.*"

Stephen Hawking

A web service is a logically atomic component that provides operations through a standardized interface and is accessible on the Internet. The most important features of a web services platform is the use of the Internet high-level protocols for making software components interact among each other. Web services use three major standard technologies, namely: a communication protocol (SOAP - *Simple Object Access Protocol)* [23]), an interface description language for describing services (WSDL - *Web Service Definition Language* [30]) and a specification for publishing and finding services (UDDI - *Universal Description, Discovery and Integration* [54]). These technologies are organized within a reference architecture that defines how they are used for using services as components that can exchange messages for implementing specific application logics.

Furthermore, it is possible to create new services (*i.e., composite services*) by composing existing ones. Service composition is defined by a process consisting of activities implemented by other Web services.

Some composition processes require to define more complex interactions among services. For this, it is necessary an adequate description of the interface that can describe composite services. Since WSDL descriptions specify only static aspects, some proposals extend WSDL adding behavioral characteristics, such as *PEWS [27], BPEL4WS [9], XLANG [87], WSCI [10]* e *OWL-S [17]*.

Current standards in service composition implement functional, non-functional constraints and communication aspects by combining different languages and protocols. For example, to add a transactional behaviour to a service composition requires the implementation of code using the WS-Coordination, WS-Transaction, WS-BussinessActivity and WS-AtomicTransaction standards. The selection of the adequate protocols for adding a specific non-functional constraint to a service composition (e.g., security, transactional behaviour and adaptability) is completely left to the programmer. As a consequence, the development of an application based on a service composition is a complex and a time-consuming task. This is opposed to the philosophy of service-based software engineering that aims at facilitating the integration of distributed applications.

In general, software engineering techniques have been conceived for supporting the development of applications, and service-based applications should not be the exception. The evolution of the software engineering field has led to the proposal of the MDA (Model Driven Architecture) approach. MDA [67] is an important approach for the alignment between high-level information modelling, non-functional requirements and service-based development, because it provides a conceptual structure that extends from the models used by system analysts to different models used by software developers.

MDA allows the specification of a system as an abstract model, which may be realized as a concrete implementation (program) for a particular service platform (e.g. WSDL, BPEL or PEWS). Using a MDA specification it is possible to generate the source code for implementing the software system from an abstract model. The MDA methods provide a skeleton of the program source code, implemented as a source code template where predefined tokens are then replaced with code. MDA methods separate design from architecture. The design addresses the functional requirements while architecture provides the non-functional requirements like scalability, reliability and performance are realized.

## 1.1 Motivation and Problem Statement

Software construction is moving towards the use of methodologies for controlling the software development process and facilitating the specification of functional and non-functional requirements (NFRs), and its maintenance and evolution for integrating new functionalities. There exist several methodologies, methods and new approaches [37, 77, 85, 46, 20, 66] for software engineering, improving software development and modeling non-functional requirements. These methods can help to ensure the coherence between functional and non-functional properties necessary to the system, particularly when information systems include complex business processes calling web services or legacy applications exported as services. The challenge is to ensure reliability properties for the complete application. There are still a number of problems to be solved in the development of systems, specially for service-based development, *e.g.*, automatic composition, NFRs specification and modelling, or specific development methodologies for service-based applications.

Service oriented computing is at the origin of an evolution in the field of software development. An important challenge of service oriented development is to ensure the alignment between IT systems and the business logic. Thus, organizations are exploring new mechanisms to bridge the gap between the developed systems and business needs [19]. The literature stresses the need for methodologies and techniques for service oriented analysis and design, claiming that they are the cornerstone in the development of meaningful service-based applications [74]. In this context, some authors argue that the convergence of model-driven software development, service orientation and better techniques for documenting and improving business processes are the key to make real the idea of rapid, accurate development of software that serves, rather than dictates, software users' goals [94].

Service oriented development methodologies providing models, best practices, and reference architectures to build service based applications mainly address functional aspects [3, 4, 38, 77]. Non-functional aspects concerning service and application's "semantics", often expressed as requirements and constraints in general purpose methodologies, are not fully considered or they are added once the application has been implemented in order to ensure some level of reliability (e.g., data privacy, exception handling, atomicity, data persistence). This leads to service based applications that are partially specified and that are thereby partially compliant with application requirements.

The use of many different platforms and APIs for the development of Web applications raises the need for an infrastructure that allows an organized execution of the analysis activities, design and implementation, enabling the design, implementation and development of reliable applications.

As services are independent components, ensuring non-functional properties is a challenge. Different studies [14, 7, 29, 50, 95, 56, 90] try to associate non-functional requirements to services using different approaches. Associating non-functional requirements to services compositions can help to ensure that the resulting application is compliant to the user requirements and also with the characteristics of the services it uses.

Our approach combines service-based development, non-functional aspects and model driven architecture (MDA) so that reliable service-based systems may be developed by reducing time delays and increasing results, such as compliance and quality.

## 1.2 Objectives and Main Results

The main goal of this thesis is to provide a methodology for developing reliable web service applications. Addressing the (i) modelling of functional requirements that can be implemented by service compositions and (ii) non-functional requirements expressed as constraints associated to services and policies to ensure the enforcement of business rules that define the semantics of the application.

Our work proposes the MDA based methodology $\pi$SOD-M (*Policy based Software Oriented Development Methodology*) that (i) extends the SOD-M [38] method with the notion of *Policy* [55] for representing non-functional constraints associated to service based applications; (ii) defines the $\pi$-*PEWS* meta-model [80] providing guidelines for expressing the composition and the Policies; and finally, (iii) defines model to model transformation rules for generating the $\pi$-*PEWS* model of a reliable service composition starting from the extended service composition model; and, model to text transformations for generating the corresponding implementation.

This work proposes also an environment that provides tools for using the methodology and implementing reliable services based applications. By defining the $\pi$SOD-M model a programmer can design and implement cross-cutting aspects (e.g., exception handling for describing what to do when a service is not available, recovery, persistence aspects) and constraints associated to services, that must be respected for using them (e.g., the fact that a service requires an authentication protocol for executing a method).

The main contributions of our work are:

- **Classification of non-functional requirements for web services:** This classification is based on an analysis of the key features in the development of service-based applications. Associating non-functional requirements to a service composition partially ensures that the resulting application will comply to the user requirements as well as with the characteristics of services user for implementing it.
- **The definition of non-functional requirements meta-models:** $\pi$SOD-M defines non-functional requirements meta-models for representing concepts associated to services properties and constraints, and business rules and thereby providing a framework for modelling reliable service-based applications.
- **MDA based methodology for modelling and implementing reliable service-based applications:** $\pi$SOD-M defines a set of concepts organized according to three points of

view: (*i*) *policy view*, focusing on the non-functional requirements expressed by policies, (*ii*) *business view*, focusing on the features and the requirements of the business, and (*iii*) the *system view*, concentrating on the functionalities and processes that need to be implemented in order to satisfy the business requirements. The modelling process of $\pi$SOD-M includes models that are in correspondence with the three different abstraction levels considered by MDA (CIM, PIM and PSM).

- **$\pi$SOD-M environment:** We the $\pi$SOD-M environment implemented as Eclipse plugins that enables the use of the proposed methodology for specifying functional and non-functional requirements of a reliable service-based application and for semi-automatizing the transformation processes from CIM to PIM and PIM to PSM levels, and then generating executable code.

## 1.3 Document Organization

The remainder of this document is organized as follows:

- Chapter 2 presents a systematic review of approaches associating non-functional properties to services and service-based applications. It introduces the main concepts used for describing and modeling non-functional requirements in the different phases of service-based methodologies;
- Chapter 3 introduces the $\pi$SOD-M methodology proposed in this thesis, used for building reliable service-based applications. The chapter describes the main phases of the methodology which is based on an MDA approach, and the meta-models associated to the CIM, PIM and PSM levels. Developing a service-based system consists ins instantiating such meta-models and transforming them along the different levels. This chapter also describes the transformation rules among the meta-models of the $\pi$SOD-M methodology, that includes model-to-model and model-to-text transformations.
- Chapter 4 describes the $\pi$SOD-M environment for developing service-based systems using this methodology. The chapter describes the architecture of the environment, its main components and functions and the tools for implementing it;
- Chapter 5 describes the experimental scenarios that we conducted for validate the $\pi$SOD-M methodology. The chapter introduces three scenarios with different levels of complexity, some focusing on complex application logics and few non-functional requirements,

and others having in contrast simple application logics, but complex non-functional requirements. The $\pi$SOD-M methodology is systematically used for developing the case studies. Then, a discussion is conducted for presenting lessons learned and for comparing this methodology with a methodology of reference.

- Chapter 6 concludes this document. It enumerates the main results of this work underlining the main contributions and then it discusses future perspectives.
- Some details about our work are described in the appendices:
  - Appendix A describes the method used for source selection and analysis of related works with non-functional requirements for service-based applications;
  - Appendix B presents a list of service-based non-functional requirement concepts and definitions proposed in this thesis;
  - Appendix C presents the $\pi$-*PEWS* language used in this work.

# 2 State of the Art

"*Computer science is no more about computers than astronomy is about telescopes.*"

Edsger W. Dijkstra

Service oriented development methodologies providing models, best practices, and reference architectures to build service-based applications mainly address functional aspects [85]. Non-functional aspects concerning services and applications semantics are often expressed as requirements and constraints. It is common that these aspects are not fully considered during the development of applications. In many cases, they are considered once the application has been implemented, in order to ensure some level of reliability (e.g., data privacy, exception handling, atomicity, data persistence). This situation leads to service-based applications that are partially specified and that are partially compliant with initial application requirements.

In systems and requirements engineering, a non-functional requirement (NFR) specifies criteria about the behaviour of a system. These criteria may be not related to the results produced by the system, but to other conditions of its performance and execution. Non-functional requirements are often called qualities of a system. They are also referred as "constraints", "quality attributes", "quality goals", "quality of service requirements" and "non-behavioural requirements" [86]. In the case of service-based applications, non-functional requirements concern the application itself as well as its component services.

In the case of service-based applications, non-functional requirements concern the application itself, as well as constraints imposed by the services. Associating non-functional requirements to services composition can help to ensure that the resulting application is compliant to the user requirements and also with the characteristics of the services it uses. As services are independent components, ensuring non-functional properties is a challenge.

Programming non-functional properties is not an easy task and different studies [14, 7, 29, 50, 95, 56, 90, 59] associate non-functional requirements and services using different approaches. Although they are used as synonyms in most NFR approaches [59], we distinguish the concepts of non-functional requirements and non-functional attributes. NFRs are defined as a group of semantically correlated non-functional attributes (NFA). For example, *security* is an non-functional requirement that comprises attributes such as *confidentiality* and *integrity*. A non-functional attribute describes the characteristics of a functional requirement.

This chapter *(i)* presents the state of the art of web service based methodologies and non-functional requirements for web service based applications, and *(ii)* analyses the principal methodological concepts for web service development. The bibliography will be analyzed in order to present the different existing concepts, proposals, problems and solutions related with each area.

The analysis will be conducted through a systematic review[1] [57]. The result of this review can be useful to summarize the existing evidence concerning a treatment or technology, to identify gaps in current research in order to suggest areas for further investigation, and to provide a background for new research activities.

We present an analysis organized in 2 parts to facilitate the understanding of the areas and related works and what their importance in the context of service-based development. Our goal is to identify a common nomenclature for existing NFR and web service based methodologies and propose a specification of these concepts in the context of reliable web services development.

Since the focus of this work is the development of reliable service-based applications, we analyze *what*,*where* and *how* non-functional requirements are modeled in the development of this type of application. Allied to this, we analyze which methodologies address development of service-based applications and if they model non-functional aspects.

This chapter is organized as follows. Section 2.1 presents the main concepts and works related with non-functional requirements for web service application. Section 2.2 presents the main concepts and works related with methodologies for web service development. Section 2.3 present a non-functional requirements model for service-based development and a classification of the main terms used, divided into levels. Section 2.4 concludes for this chapter.

[1]A systematic review provides means of identifying, evaluating, and interpreting the literature relevant to a particular research question or topic area [57].

## 2.1 Non-Functional Requirements for Service-Based Applications

This section presents a systematic review of the non-functional requirements and properties used in the context of service-oriented development. The purpose of our analysis is:

- to identify the concepts, properties and notations used in the service-based systems development;
- to find if there is any pattern or relationship between non-functional requirement concepts in different modelling levels.

The functional and non-functional requirements are refined in each phase of the development. As a result, the granularity of the requirement becomes tiner and more precise.

We propose 7 research questions ($RQ_1$ to $RQ_7$) to guide our analysis of the bibliography about non-functional requirements. For each question we define a set of possible answers in order to guide the analysis. These possible answers are defined from our knowledge on each topic. The questions are closely related to service-oriented development with a NFR focus. They are:

- $\boldsymbol{RQ_1}$ **:** How are NFR modelled by existing methodologies for developing reliable Web services?
  - *Answer is specific to each proposal*
- $\boldsymbol{RQ_2}$ **:** Which are the NFR that are more frequently used by methodologies developing web services?
  - *Possible answers: security / availability / portability / . . . / reliability / performance*
- $\boldsymbol{RQ_3}$ **:** What is the software development approach used in the work?
  - Possible answers: *Model driven approach (*MDD) / Ontology (*Ont) / Formal method (*FM) / Artificial intelligence (*AI) / Business Process Modeling (*BP) Traditional (*TDT)*
- $\boldsymbol{RQ_4}$ **:** What is the discipline (application domain) of the "*non-functional requirements*" / "*non-functional properties*" used in the work?

  - Possible answers: *Software architecture / QoS model / Language definition / Methodology / etc*

- $\boldsymbol{RQ_5}$ : Does the paper propose a (meta)model describing and analyzing NFR? Is there any relationship between the proposed non-functional requirements (meta)model and business services?
  - Possible answers: *yes / no – yes / no*

- $\boldsymbol{RQ_6}$ : Are the non-functional aspects treated in an independent way?
  - Possible answers: *single / composition*

- $\boldsymbol{RQ_7}$ : Which is the publication year of the paper?
  - Possible answers: *Year of publication*

The research questions identify characteristics of the service-oriented application development, especially the modelling of non-functional requirements/properties, how they are addressed and if they are classified.

### 2.1.1 Concepts and Works

In order to classify the available works using the research questions, we proceed with a systematic selection of the published papers in our field of interest.

We used the approach described in Appendix A for searching, collecting and selecting some works related with non-functional requirements for developing service-based applications. This approach selects published articles from known sources to be considered for inclusion in our study. Based on these data we analyzed concepts used in each work for the description of NFRs and the differences between them. The notation used to refer non-functional requirements will be highlighted in *italic*, including the set of values that can be associated to each concept. For example, a notation used by a work can be *quality property* or *non-functional concern*, and the values associated with its notation are *security*, *reliability*, *transaction*, etc. Thus, considering each research question, we analyze a set of works in order to identify any pattern or divergence between the analyzed literature.

Babamir et al.[14] ranks services *quality properties* in three categories (business level, service level, system level). *Quality properties* are associated with *quality constraints*, that are defined as assertions or propositional logic formulas. *Non-functional attributes, composition model entity* and *model entity* are the notations used by Xiao et al. [95] for classifying the different concepts for non-functional requirements modeling. Non-functional attributes (NFAs) describe the non-function aspects in the abstract process models. The framework to model NFRs proposes to annotate composition models with NFAs.

D'Ambrogio [34] uses the term *quality characteristics* to group similar characteristics into *quality categories*. Each *quality characteristic* is quantified by *quality dimensions*. *Quality characteristic* is a quantified QoS aspect, for example *latency, throughput, reliability, availability*, etc. *Quality characteristics* of a common subject are grouped into abstract *quality categories*, for example *performance* (for latency and throughput characteristics) and *dependability* (for reliability and availability characteristics).

The terms *category, sub-category* and *property* are used by Yeom et al.[96] to classify non-functional requirements. *Business*, *service* and *system* are the possible values to be associated with the proposed terms, and a *sub-category* can be *security, value, interoperability*, etc. The work in [96] defines a *web services quality model*, which considers non-functional properties in several aspects. In this model, web services qualities are classified in categories and sub-categories. Chollet et al.[29] uses only two terms to classify and relate quality properties with services, they are: *activity* and *quality property*. Each activity represents a functional property that can be divided in sub-activities, depending on its granularity. For non-functional requirements, the work in [29] describes the possibility of creating different meta-models for each *quality property*, and then relate them with activities.

Schmeling et al.[84] uses the *non-functional concerns (NFC)* notation to describe NFRs. This term encompasses two aspects: the specification of NFCs and their realization. [84] defines that a *functional concern (FC)* is a consistent piece of functionality that is part of a software system. *Non-functional concern (NFC)* is a general term describing a matter of interest or importance that does correspond to a non-functional requirement of a system, for example, *security, reliability, transactional behavior*, etc. A *non-functional action* represents some behavior that results in a *non-functional attribute*. An example of *non-functional action* is *encryption*, which realizes the *non-functional attribute*, *confidentiality*. A *non-functional activity* is also used as a term, which means to encapsulate the control flow of *non-functional action* that apply to the same subject. The term is used in analogy to activity and action in UML2.

Ceri et al.[26] uses the terms *policy, rule, condition* and *action model* to specify NFRs, and in a similar way, Agarwal et al.[7] also uses the concept of *service policy* associated with the concept of service. Each service is also associated with a *service property*, which may have a specific value (*security, reliability*, and so on). Each service is also associated with a *unit function*, that represents one or more requirements.

Ovaska et al.[73] uses *quality attribute, category, conceptual layer* and *importance* to organize and classify the NFRs, Pastrana et al.[78] uses the term *contract* to describe non-functional requirements. In a *contract* it is possible to define pre-conditions, post-conditions and invariants. A *web service* can have many *contracts*, that defines many *assertions* and are associated with *quality properties*.

Some related papers do not have any nomenclature to classify non-functional requirements. However, some of them uses the *attribute* notation [97, 16, 56], other use *properties* [43], *factors* [69, 50], *characteristics* [41], *quality level* [68] or *values* [88, 16].

### 2.1.2 Analysis

Although there exist many different types of notation used for classifying non-functional requirements, in general, the values associated to them are the same, *i.e. security, performance, reliability, usability, availability*, etc. What distinguishes the different approaches is the adoption of different NFR hierarchies in order to prioritize or classify the quality requirements. Another interesting factor is that each work uses different approaches to model these requirements. Thus, there are different kinds of notations for non-functional requirements.

A number of approaches use [34, 29, 84, 16, 43, 73] MDD (Model Driven Development) for designing and developing systems. Fabra *et al.* [43] presents a complete methodology that does not focus on non-functional requirements. Fabra *et al.* [43] describes the importance of the use of MDD in the development of service-oriented applications. [88, 97] use formal methods to define a development process based on NFR for web services, [7, 78] use ontologies for the definitions and modeling of non-functional requirements and [95, 50] use Business Process Modeling (BPM) for system specification, including NFR. Most works focus on composition service modeling while others define requirements models for representing the properties they use.

In [14, 96] non-functional properties for web services are classified into three views such as *service level, system level* and *business level*. In the *business* level the quality properties are: *service charge, compensation rate, penalty rate* and *reputation*; at the *service* level the quality properties are: *performance* and *stability*; and at the *system* level the quality properties are classified into: *manageability, interoperability, business processing* and *security*.

In the method defined in [95], each task in the process model is annotated with the NFAs (*non-functional attributes*). During the design phase, the service composition and the definition of NFAs are separated. Then, each task in the process model is annotated with the corresponding NFA. The attributes are related with tasks or data item. For data, the NFAs are: *value* and *range*; and for tasks the NFAs are: *cost, time, resources* and *expression.*

D'Ambrogio [34] presents a WSDL extension for describing the QoS of web services. The process is based on MDA. The work presents a catalog of *QoS characteristics* for the web service domain and the Q-WSDL (Quality WSDL) meta-model for modeling QoS properties in web services. The properties presented are: *availability, reliability* and *access control.* [29] presents a security meta-model for web service composition. The non-functional requirements considered are *authentication, integrity* and *confidentiality*. Each property is related with a service activity. In [84], authors present an approach and a toolset for specifying and implementing the composition of several non-functional properties of Web services. The non-functional attributes described in [84] are: *confidentiality, integrity (security concern)* and *response time (performance concern).*

The work presented in [88] describes steps to design a selection mechanism of services identified as candidates for participating in a composition, considering their quality properties. The steps are: *(i)* identification of relevant QoS information; *(ii)* identification of basic composition patterns and QoS aggregation rules for these patterns; and *(iii)* definition of a selection mechanism of service candidates. As QoS properties considered in [88] are: *performance, cost, reliability* and *availability*.

Among the properties presented we highlight *security* and *performance*. Users usually need to access data securely and quickly. Most studies have both properties as the most important non-functional requirements. *Reliability* is also an important non-functional requirement presented in some works and required by end users.

We can now relate the research questions presented in section 2.1, with the works described above. Tables 1 and 2 show the results.

The vocabulary used for naming and characterizing NFR is not stable, 5 works out 360 (1.6%). 19 of the total of works chosen (26.31%) propose a classification of non-functional requirements (see appendix A).

There are other works [32, 33, 31, 47, 85] that consider and propose non-functional requirements classifications. A number of software attributes are defined as requirements in [85, 32]. According to [32], the most common non-functional properties are: *performance*; interface; operational; resource; verification; acceptance; *maintainability*; documentation; *se-*

| **Reference** | $RQ_1$ : ***NFR concepts*** | $RQ_2$ : ***NFR values*** | $RQ_3$ : ***Approach*** | $RQ_4$ : ***Domain / Scope*** |
|---|---|---|---|---|
| Babamir et al. [14] | property / category / constraint | responsiveness / availability performance / sla properties | TDT | Software architecture |
| Yeom et al. [96] | category / sub-category / property | business value / performance / stability /manageability / security / business processing interoperability | TDT | QoS model |
| Xiao et al. [95] | NF attribute | time / cost / resource | BP | Business processes modeling |
| D'Ambrogio [34] | characteristics / category / dimension | availability / reliability / access control | MDD | WSDL extension |
| Chollet et al. [29] | activity / NF attribute | security | MDD | Orchestration Framework |
| Schmeling et al. [84] | NF concern / NF attribute / NF action / NF activity | security | MDD | Web service composition process |
| Thißen et al. [88] | NF value | performance / reliability cost / availability | FM | Software architecture |
| Zhang et al. [97] | attribute / predicate | security | FM | Access control |
| Basin et al. [16] | attribute | security | MDD | System architecture |
| Ceri et al. [26] | police / rule condition / action | n.a. | TDT | Context-aware applications |
| Fabra et al. [43] | property | storage / processing (*case study) | MDD | Web service methodology |
| Modica et al. [68] | quality level | sla properties | TDT | Service oriented architecture |
| Ovaska et al. [73] | attribute category | security / reliability | MDD | Model development |
| Agarwa et al. [7] | property / policy / function | *not explicitly defined* | Ont | Policy language |
| Jeong et al. [56] | NF attribute | operation cost / performance / availability / accessibility / security / interoperability / usability / user satisfaction | AI | Service oriented architecture |
| Pastrana et al. [78] | NF property / contract / assertion / NF behaviour | performance / reliability / scalability / capacity / robustness / precision / accessibility / availability / interoperability / security | Ont | Web service methodology |
| Diamadopoulou et al. [41] | NF characteristic | user' subjective perception | TDT | Web service selection |
| Gutierrez et al. [50] | NF factor NF sub-factor | Security | BP | Web service development process |
| Mohanty et al. [69] | NF attribute or NF factor | reliability / performance / integrity / usability response time / documentation | AI | Artificial intelligent / Web services classification |

Table 1: Research question results - $RQ_1$, $RQ_2$, $RQ_3$, $RQ_4$.

*curity*; *portability*; quality; *reliability*; *usability*; and *safety*. We *highlight* those that are used in most analyzed classifications. [32] proposes an NFR specification and a template for specifying requirements, considering both, functional and non-functional properties.

The NFRs are classified in 4 main concepts by [33]. The classification uses *quality properties*. These properties are: *performance; security; cost;* and *usability*. Pastrana [78] describes an ontology based methodology and uses a NFR classification based in some properties, most already described in other works. However that work is the only one to use *scalability, capac-*

| **Reference** | $RQ_5$ : ***Service model – Business services*** | $RQ_6$ : ***Service type*** | $RQ_7$ : ***Year of publication*** |
|---|---|---|---|
| Babamir et al. [14] | no – yes | composition | 2010 |
| Yeom et al. [96] | yes – yes | single | 2006 |
| Xiao et al. [95] | no – no | composition | 2008 |
| D'Ambrogio [34] | yes – no | composition | 2006 |
| Chollet et al. [29] | yes – yes | composition | 2009 |
| Schmeling et al. [84] | no – no | composition | 2011 |
| Thißen et al. [88] | no – yes | composition | 2006 |
| Zhang et al. [97] | no – no | single | 2005 |
| Basin et al. [16] | yes – no | single / composition | 2006 |
| Ceri et al. [26] | no – no | single | 2007 |
| Fabra et al. [43] | yes – yes | composition | 2011 |
| Modica et al. [68] | no – no | composition | 2009 |
| Ovaska et al. [73] | yes – no | single | 2010 |
| Agarwa et al. [7] | yes – no | single / composition | 2009 |
| Jeong et al. [56] | no – no | composition | 2009 |
| Pastrana et al. [78] | yes – no | composition | 2011 |
| Diamadopoulou et al. [41] | no – no | composition | 2008 |
| Gutierrez et al. [50] | no – no | single / composition | 2010 |
| Mohanty et al. [69] | no – no | single | 2010 |

Table 2: Research question results - $RQ_5$, $RQ_6$, $RQ_7$.

*ity* and *precision* properties. Considering web services development, these properties are not very frequently used, however, considering data processing, these properties can be important in cases of large data processing requests through services.

Sommerville [85] classifies requirements, either functional or non-functional, in three main blocks: *process, product* and *external*. [85] also defines a sub-classification considering the system domain. Software requirements are classified as: *usability; reliability; safety; efficiency; performance;* and *capacity*.

Some requirements may determine the system design, for example, *(i)* keep certain functions in separate modules; *(ii)* check data integrity for critical variables; and *(iii)* permit only limited communication (requester / provider), are examples of restrictive requirements [32]. These examples are related with some NFR concepts as: *security, reliability* and *availability*.

In the service-based development there is a clear difference between the business, service and system levels. Quality requirements are treated differently in each of these levels. Despite the different nomenclatures, they can be described in general as *non-functional concerns/requirements* and *non-functional attributes*.

Restrictions are usually related to use cases and functional requirements ([32] and [85]), and in most cases, a service activity can be represented as an use case. This is the way in which non-functional requirements are related with web services. They do not propose a specific way for designing services constraints, they assume that a service is modeled as a use case, and it has quality requirements.

At the service level, each service activity is related in some way with the concept of contract.

Contracts can be grouped into policies and their rules. Policies are directly related to concepts like quality, security, performance and availability; contracts are associated with non-functional attributes.

Due to the variety of tools and new approaches for web service development, there is not (yet) a consensus on a software process methodology for web services [77, 75, 63, 66, 44, 82, 45].

Works proposing methodologies can be classified into two types: (*i*) those that propose new approaches for non-functional properties guarantees; and (*ii*) those proposing service-oriented development methodologies for the whole development process.

[52] proposes *Design by Contract* for web services. It is possible to describe three levels for specifying contracts: *implementation level, XML level* and *model level.* Design by Contract applied to web services addresses the verification of web services through runtime checkers, before the deployment, such as *jmlrac* [58], by adding behavioral information to the specification of services, using JML [58].

CDL (*Contract Definition Language*) [63] is a XML-based description language, for describing contracts for services. The development with CDL offers an architecture framework, design standards and a methodology [65, 62, 64, 66], that can be easily understood and applied to the development of applications. The greatest difficulty is that the language only represents contracts for services. Its specification is generated by refining B [6] machines that describe the services and their compositions.

## 2.2 Methodologies for Service Oriented Development

In this section, we present an analysis of methodologies for service-oriented development. We defined 5 research questions to guide our analysis. The questions are closely related to service-based systems. The criteria used for evaluation and comparison are:

- $\boldsymbol{RQ_8}$: What is the scope of the methodology?
  - *This item will review the major phases and the purpose of each particular phase in the context of the project and service-oriented development.*
- $\boldsymbol{RQ_9}$: How does the methodology define the development process according to the adopted notation?

  - *This item describes which notation is used by the methodology for designing its models, e.g., MDA, UML, SysML, etc.*

- $\boldsymbol{RQ_{10}}$ **:** Does the methodology use a formalism? If so, What is the formalism used for specifying services?
  - *The purpose of this item is to determine whether the methodology proposes the use of a formalism for specifying the services, their compositions and iterations.*
- $\boldsymbol{RQ_{11}}$ **:** What is the approach for describing services?
  - *The purpose of this item is to determine the approach adopted by the methodology for describing the development process.*
- $\boldsymbol{RQ_{12}}$ **:** Which models does the methodology propose?
  - *This item discusses the models proposed by methodologies for specifying the system behavior and modeling high-level business requirements.*

Along with the research questions, we also compared the MDA models proposed by different methodologies.

In general, for service-based systems, there is no specific method to conduct the development process. Our analysis, helped us to identify how these methodologies define activities and models to design reliable services, as well as service-based applications.

### 2.2.1 Concepts and Works

Over the last few years, a number of approaches have been proposed for the development of web services. These approaches range from the proposal of new languages for web service descriptions [9, 71, 15, 60, 83] to techniques to support phases of the development cycle of this kind of software [22, 21]. In general, the objective of these approaches is to solve specific problems, like supporting transactions or QoS, in order to improve the security and reliability service-based applications. Some proposals address service composition: workflow definition [92, 70] or semantic equivalence between services [13]. The proposed solutions come from many communities, including those of Theoretical Computer Science [83, 93, 51, 35, 25], Software Engineering [24, 92, 91, 12, 61], Programming Languages [71, 9] and Databases [79, 5]. However, in spite of the variety of tools, there is not (yet) a consensus on a software process methodology for web services.

There are methodologies that address the service-based development towards a standard or a new way to develop reliable service-based applications. The methodologies analyzed in this work are: SOD-M [36] and SOMF [20] representing the model based development for web services; S-Cube [76] representing the business processes and service-based development; SOMA [11] that is a methodology described by IBM for SOA solutions; Extended SOA [77] merges RUP[1] and BPM[2] concepts for service modeling; DEVISE [40] is a methodology for building service-based infrastructure for collaborative enterprises. Furthermore, there are other proposals, such as the WIED model [89] that acts as a bridge between business modeling and design models. Also, traditional approaches for software engineering [85] are being applied to service development.

**SOD-M:** Service-Oriented Development Method [37] proposes a service-oriented approach and model-based development for web systems. SOD-M proposes models and standards primarily focusing on the development of the systems behavioral characteristics, setting standards for the construction of business models at a high-level of abstraction. The approach describes three meta-models organized according to the MDA (*Model Driven Architecture*) [67] architecture levels: CIM (*Computational Independent Models*), PIM (*Platform Independent Models*) and PSM (*Platform Specific Models*).

At the CIM level, 2 models are defined: *value model* [48] and *BPMN model*; At the PIM and PSM levels, DSL models for service. The PIM-level models are: *use case, extended use case, service process* and *service composition*. The PSM level models are: *web service interface*, *extended composition service* and *business logic*.

The methodology provides a framework for the development of service-oriented applications with models that can express the whole development process of services-based applications. However, SOD-M has no support for describing and modelling non-functional requirements.

The basis of SOD-M is a set of concepts guiding the whole development, including transformations between models. The concepts are represented through a meta-model. This meta-model describes concepts of both the business and system views.

In SOD-M, the PIM level models the entire structure of the application flow, while, the PSM level provides transformations towards more specific platforms.

The methodology provides model transformations among: *CIM-to-PIM, PIM-to-PIM* and *PIM-to-PSM* transformations. Given an abstract model at the CIM level, it is possible to apply transformations for generating a model of the PSM level. In this context, it is necessary to follow the process activities described by the methodology. The model to model transformations are defined using ATL [49].

SOD-M defines a set of concepts according to two points of view: *(i) business*, focusing on the characteristics and requirements of the organization and *(ii) system requirements*, focusing on features and processes to be implemented in order application requirements. In this way, SOD-M aims to simplify the design of service-oriented applications, as well as its implementation using current technologies.

**S-Cube:** The S-Cube Framework [76] proposes an integrated structure for developing service-based applications. S-Cube offers three areas: *Business Process Management, Composition and Coordination of Services; and Infrastructure*. These areas are the backbone of the framework that are directly linked to three other areas for supporting systems development: *Engineering and Software Design; Monitoring; and Security and Software Quality*.

The methodology aims to guide the development of applications and describes some essential activities, such as *(i) description of business objectives*, *(ii) domain assumptions defining*, which are pre-conditions to be met for a particular application domain, *(iii) description of domains*, and *(iv) description of scenarios for each domain*.

The S-Cube methodology does not provide an exhaustive list of rules for describing services. The S-Cube framework provides activities in various service-oriented development areas, however, it is still required to apply its concepts in real case studies to give an idea of its application, given the fact that its structure is very complex and multidisciplinary.

**DEVISE:** [40] identifies issues to be considered in service-based design, and gives generic guidelines for addressing them. DEVISE helps the developer in the design of new applications as a collection of web services, and provides means of porting existing applications to services-based platforms.

**SOMA:** *Service-Oriented Modeling and Architecture* [11] is a methodology by IBM for SOA solutions. SOMA defines a seven-phase development life-cycle: *business modeling and transformation; management solution, identification, specification, realization, implementation*, and *monitoring of implementation and management*.

At each stage, different tasks are carefully defined, such as the definition of business architecture, development of service model, specification of services, etc.

The SOMA conceptual model is based on the SOA architectural style it defines an interaction model with three main parts: (i) *the service provider* (which publishes a service description and provides the implementation for the service); (ii) *consumer services* (which can use the URI for the service description directly or can find the service description in a service registry for the call); and the (iii) *service broker* (which provides and maintains a record of services).

SOMA proposes the use of IBM Rational platform for SOA developing, and other different IBM tools that are available to analysts, software architects and application developers.

**SOMF:** *Service-Oriented Modeling Framework* [20] is a model oriented methodology for modeling software with the best practices of software project activities and different architectural setting. SOMF can be used to describe enterprise architectures, service-oriented architecture (SOA) and cloud computing. SOMF offers a variety of modeling practices and disciplines that can contribute to developing a successful service-oriented applications. The main goals of SOMF are [20]:

1. A methodology describes SOMF modeling activities and each model transformation;
2. The diagrams are created obeying some project patterns.

The methodology's model transformations in SOMF aim to describe and refine aspects in the system development process. The models are: *discovery model, analysis model, design model, architectural model, implementation model, quality model, operations model, business model, governance model.*

**Extended-SOA:** [77] adopts the lifecycle of web development services focussing on the *analysis, design* and *development* phases, with approaches that are centered in functional requirements. According to [77], a specific methodology for developing services-based is important to specify, build, improve and customize business processes available on the Web.

Extended SOA has a hierarchical structure for the development of web services, where layers are defined as follows: *Business (Service) Domain, Business Processes, Business Services, Infrastructure Services, Component-based Service Realizations* and *Operational Systems.*

The methodology is partly based on other successful models and related development, such as the Rational Unified Process (RUP), Components-based development and Business Process Modeling (BPM). The difference with those models is that Extended SOA life cycle focuses specifically on service-oriented development.

Extended SOA is based on three principles for the design and development of services, they are: *Service coupling, Cohesion of service* and *Service granularity.*

Considering service-oriented projects, it is preferable to create high-level interfaces with coarse granularity to implement a complete business process, since multiple calls of service increases network traffic. As the services are in a cooperative environment, it is unfeasible to create services with finer granularity. It is preferable to create functions with finer granularity in a local environment. Thus, the methodology aim is to achieve services integration and interoperability. The Extended SOA phases are: *Planning, Analysis, Design, Implementation, Testing, Provisioning, Implementation, Execution* and *Monitor*.

**Traditional Software Engineering Approach:** Sommerville [85] proposes a general approach for the development of applications that use web services. Its structure uses the activities of design, development, construction and testing of web services and their compositions. The proposal is based on the traditional concepts of software engineering, such as:

- Software reuse: standards-based web service that provides a mechanism for inter-organizational computing;
- Use process engineering services to produce reusable web services;
- Perform composition of services to facilitate and improve the development of applications;
- Show how business process models can be used for the design of service-oriented systems.

Service engineering allows the development of reliable and reusable services. Thus the entire development life cycle focusses on the reuse of independent services.

A service must be designed as a abstraction that can be reused by different systems. The main activities for service engineering are: *(i)* Identification of candidate services; *(ii)* Service design; and *(iii)* Service implementation.

### 2.2.2 Analysis

Our analysis is based on the research questions 8 to 12 described at the beginning of this section. Table 3 summarizes of the main features of the methodologies described above, considering the research questions.

S-Cube and SOMF provide concepts for modeling non-functional requirements, however they have specificities that hinder the development. SOMF proposes a notation for modeling

Table 3: Methodologies' Analysis

| | | **Service-Based Development Methodologies** | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | **SOD-M [36]** | **S-Cube [76]** | **SOMA [11]** | **SOMF [20]** | **DEVISE [40]** | **Extended SOA [77]** | **Traditional Method [85]** |
| **Methodology's Scope** | | *complete* | *complete* | *complete* | *analyse, design* | *complete* | *complete* | *analyse, design implementation* |
| **Methodology Models** | | *e-value, BPMN, UML* | *UML* | *UML* | *own notation* | *UML* | *BPEL, UML* | *UML* |
| **Formalism Used** | | *N/A* | *N/A* | *N/A* | *N/A* | *N/A* | *N/A* | *N/A* |
| **Service Specification** | | *no* | *no* | *no* | *no* | *no* | *no* | *no* |
| **Models** | **NFR** | *no* | *yes* | *no* | *yes* | *no* | *no* | *no* |
| | **Business** | *yes* | *yes* | *no* | *yes* | *yes* | *yes* | *yes* |
| | **Use Case** | *yes* | *yes* | *yes* | *yes* | *yes* | *yes* | *yes* |
| | **Service Composition** | *yes* | *yes* | *yes* | *yes* | *no* | *yes* | *yes* |
| **MDA** | **Abstract Level** | *CIM, PIM, PSM* | *N/A* | *CIM, PIM* | *N/A* | *N/A* | *CIM, PIM, PSM* | *N/A* |
| | **Model Transformation** | *yes* | *no* | *yes* | *no* | *no* | *yes* | *no* |

and does not have a environment to assist the development process, thus hindering the design of models and their transformations.

The Extended-SOA methodology and the use of traditional methods for the development of service applications seems promising, because the developer has a vision of the whole development structure. The disadvantage of these methods is that they do not provide a way to represent the service specification, such as WSDL or formalism. They also do not provide a tool for development support. Another disadvantage of traditional methods, when applied to service-based development, is the fact that these methods do not have the focus in the development of services.

In the same way S-Cube, SOD-M and SOMA provide robust processes and concepts for service-oriented development. Both methodologies have focused on model driven development. However, these methods do not provide models for representing non-functional properties. S-Cube uses UML to represent project models, covering the full development life-cycle, describing each service and their compositions. The S-Cube structure is quite extensive and complete, however, S-Cube does not provide formalism for specifying service composition.

None of the analyzed methods uses a formalism for he description and specification of services. Most of them use UML or some UML extension for model representation, and only the SOMF approach does not address the full life-cycle of development, restricting the scope in the analysis and design phases.

Considering the methodology analysis, we conclude that there are any approach that address modeling non-functional aspects as the methodology basis. Several methods propose particular features to improve the service-based development. Some proposals consider the whole life-cycle, however, do not consider non-functional aspects that should be designed when developing applications that provide and use web services.

Some development methodologies have worked with the MDA approach, defining models for the different levels of abstraction of this architecture and in some cases, such as SOD-M, DEVISE and SOMA, defining rules for automatic transformation between these models. SOD-M is a method for service-based development using MDA, and thus define models at CIM, PIM and PSM of the proposed architecture.

To conclude, based on our analysis of the data presented in table 3, an interesting methodology for reliable service-based development needs to address: *(i)* the definition of a service-oriented approach-oriented models for the development of applications, *(ii)* model non-functional properties, using UML for this, *(iii)* integrate the proposed method in the context of

MDA, providing the representation of the various applications levels, CIM, PIM and PSM, and *(iv)* provide transformation mechanism between models, focussing on quality requirements.

## 2.3 Classification of Non-Functional Requirements for Service-Based Applications

According to the previous analysis, we define the main non-functional requirement concepts that are associated with service-based development. We also present a synthesis of the concepts common to the analyzed approaches used for modelling non-functional requirements of service-based applications. The NFRs have been classified into three levels, *business, services* and *systems*. According to each modeling level, non-functional requirements are being refined from the business level to system level.

A classification of NFRs can be seen in [96]. However, in [96] NFRs are not applied on data, but to functions and service performance. It is important to classify the requirements of business and data (value) restrictions, because web services can be executed in different contexts.

In sections 2.3.1 and 2.3.2 we enumerate the concepts (NFR meta-model) and values (NFR classification) that we consider important for modeling non-functional requirements for service applications.

### 2.3.1 NFR Meta-Model

Figure 1 shows the relationship between the concepts[2] we consider important for modelling quality requirements used in service-based development.

A REQUIREMENT[3], whether functional or non-functional, can be represented by one or more use cases. A use case represents a `Service Activity`. For example, a paying process can be modeled through the use cases that represent withdrawal and deposit transactions and service provider accounts. A payment process also requires the guarantee of a complete transaction and data security. Thus several use cases model a single requirement, and can be implemented as services.

Each use case has *business* or *value* constraints. *Business constraints* are restrictions on functions and how they may be implemented. The *value constraints* are restrictions on the service interface, expressing the desired values for input and output data. Each constraint is

[2]The appendix B provides a description of each concept presented in figure 1.
[3]We use TYPEFACE to refer to the concepts of the model.

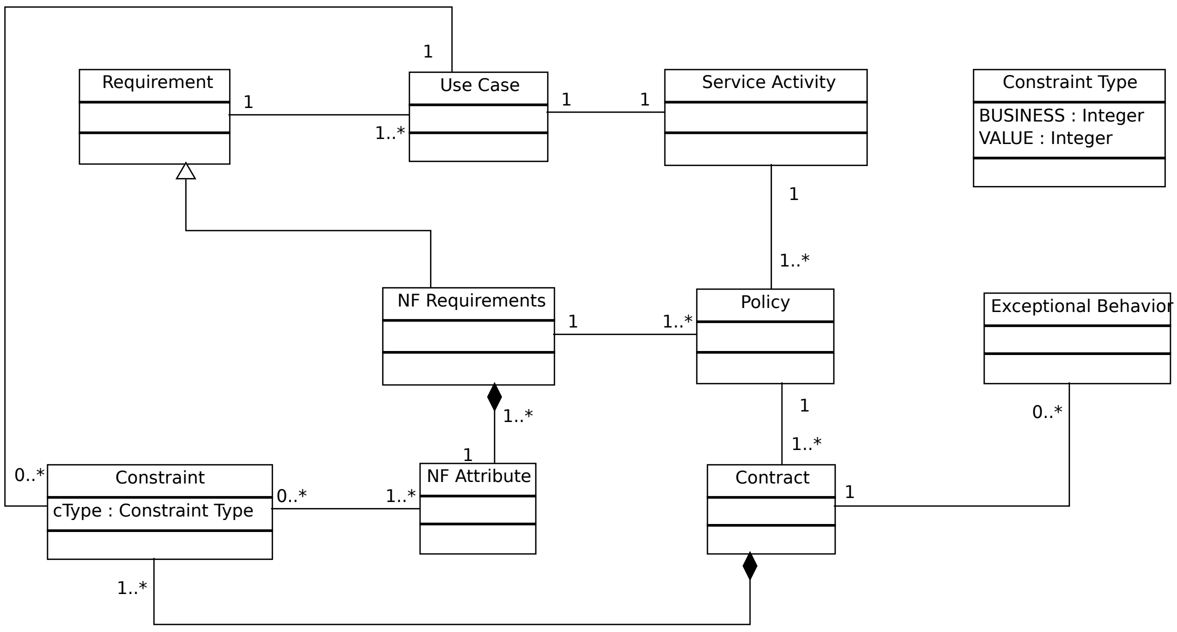


Figure 1: Service-Based Non-Functional Requirement Model.

associated with NFAs.

A CONTRACT is a set of constraints for the same function. For example, a contract for the payment operation. The constraints for payment are: (i) the value amount should not be less than 10 euros and (ii) the user should always receive a purchase confirmation by phone message. This restrictions are grouped into a single contract for payment verification.

An EXCEPTIONAL BEHAVIOR happens when a contract is not respected. When this happens a new function is called or the process is stopped. For example, if the bank does not authorize the payment, the system offers alternative forms of payment such as PayPal.

Finally a POLICY groups similar contracts. For example, security contracts are grouped into a security policy and performance contracts are grouped into a performance policy.

### 2.3.2 NFR Classification

Table 4 shows the NFR classification we propose, organized as follows (by column): (i) the level of abstraction, (ii) the proper term for this kind of abstraction and (iii) possible values to be used. The rows represent the abstraction level for modeling the NFR. The highest level is the *Business level*, the intermediary is the *Service level*; and finally the *System level*.

At the business modeling level, non-functional requirements are classified into business restrictions. We have adopted, *business* and *value* constraints to address the most abstract levels

| **Modeling Level** | ***Concept / Notation*** | ***NFR / NFA*** |
|---|---|---|
| Business Level | Constraint | Business Constraint, Value Constraint |
| Service Level | Contract | Integrity, Transaction, Accessibility, Encryption, Cost, Time Constraint, Encryption, Platform, Privacy, Authentication, Resource, Capacity, Privacy, Confidentiability |
| System Level | Policy | Security, Performance, Interoperability, Scalability, Reliability, Usability, Transactional Behaviour, Availability |

Table 4: Non-Functional Requirements Classification.

of restrictions in business modeling.

- **Business Constraint -** Represents rules for system development in terms of resource availability, interoperability, performance, dependencies, timescales. Thus, *Business Constraint* represents restrictions over business activities.
- **Value Constraint -** Defines valid subsets of values of a variable of a given type. In the context of services, value constraint are restrictions on the data on service calls (input and output values), expressing which range of values is allowed. This can be applied to authentication policies, access control, data integrity, cost of services, privacy, and other factors.

The following items describe the non-functional properties of the service level. At the service level, the non-functional properties will be guaranteed upon implementation of the service requirements.

- **Conformity -** Represents the degree at which a Web service function fulfills an application requirement.
- **Time constraint -** This property can be associated to the execution time of a particular service. This property is related to services performance and availability.
- **Capacity -** Represents the degree at which a service can process a given volume of data or requests.

- **Cost -** Represents the cost to run a service. It can be in time, bandwidth and money.
- **Privacy -** The challenge in data privacy is to share data while protecting personally identifiable information. Privacy of Web services is the ability of the service must have to preserve the data and user information.
- **Authentication -** Represents the process of making sure that the person who is asking to use the web service is really the person that they claim to be.
- **Accessibility -** Represents the quality aspect of a web service related with the degree to which it is capable of serving a request.
- **Access control -** Refers to exerting control over who can interact with a resource, *e.g.* computer-based information system, a service or a database.
- **Confidentiability -** Confidentiality ensures that information is not accessed by unauthorized people. The private data is accessed only by authorized users.
- **Integrity -** Represents the ability of the web service to maintain the correctness of the interaction which is ensured by the proper execution of a web service transaction.
- **Transaction -** A transaction is a unit of work performed to maintain a data integrity in a known and consistent state.
- **Resource -** A resource is any physical or virtual component of limited availability within a computer or information management system. Computer resources include means for receiving, processing, notifying, transmitting and storing data.

The quality properties at the system level represent a behaviour that is part of the workflow. Following, the properties we considered important for this level are defined.

- **Usability -** ISO defines usability as "*The extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use*". In the context of services, usability can mean how efficient and easy to access a service is and whether it satisfies the needs of those who are invoking this service.
- **Interoperability -** Is a property referring to the ability of diverse systems to work together. If two or more systems are capable of communicating and exchanging data, they are exhibiting syntactic interoperability. According to ISO interoperability is defined as

"*The capability to communicate, execute functions, or transfer data among various functional units in a manner that requires the user to have little or no knowledge of the unique characteristics of those units*". In the context of services, interoperability can mean that the services have capacity to communicate with each other.

- **Reliability -** Is the ability of a service or component to perform its required functions under stated conditions for a specified period of time. The concept of reliability has different applications in different areas. Applied to web services, reliability is the ability of the service to execute correctly the methods it exports. It is also important that the data presented are consistent with the expected result.
- **Availability -** Is the proportion of time a service is in a functioning condition. This is often described as a mission capable rate. Availability of web services, is the probability of a service call to be successful, given a specific time.
- **Performance -** In the context of web services, performance is how fast the service can be executed in accordance with the needs of the user.
- **Security -** Represents protecting information and information systems from unauthorized access, use, disclosure, disruption, modification, perusal, inspection, recording or destruction.

The relationship between non-functional requirements and attributes help to identify groups of restrictions and contracts to develop specific policies. The level of service shows non-functional requirements, and system level presents a finer granularity, describing the non-functional attributes. Thus, a set of contracts that are related to the same attribute form specific policies for the system. The relationship between the concepts in business, service and system levels are presented in figure 2.

In our work we adopt *security, performance, availability, interoperability, usability* and *reliability* as NFRs, and we consider as NFAs *access control, time constraint, privacy, accessibility*, *etc*. These terms and proposed values will be used in chapter 3. The methodology that will be presented in the next chapter uses this nomenclature for modeling the non-functional requirements in the development of service-oriented applications.

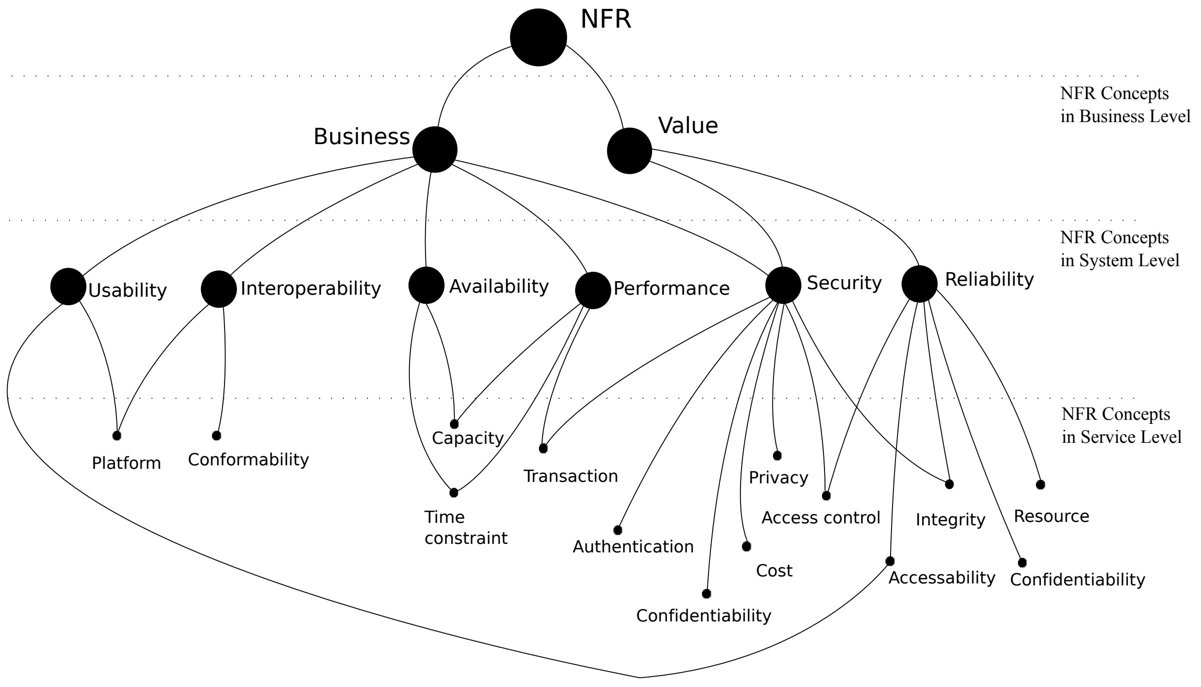


Figure 2: Relationship of the NFR Concepts.

## 2.4 Conclusions

This chapter presented a review of related approaches for the service-oriented development of applications. We consider two important topics in our analysis: *(i)* non-functional requirements and *(ii)* methodologies for service-based development. The revised methods were grouped according to their characteristics, and analyzed their context of application.

We reviewed existing methodologies for service-based development. We analyze other general aspects of these methodologies as their development paradigm, use of the UML and formalism, and whether the method considered an MDA-based development or not. In this case we have studied the main techniques and methods proposed for modelling non-functional requirements, business process and web service composition.

To the best of our knowledge, there are no proposals that define a service-oriented approach for the whole development of systems, considering non-functional requirements. Although many efforts have been made to support the new technological proposals for the Web such as web services, in general, these methodologies focus their processes on traditional software engineering approaches, with emphasis on the functional aspects of the application.

The comparative study of such methodological approaches for service development is the basis for the design decisions of the method we propose in this work.

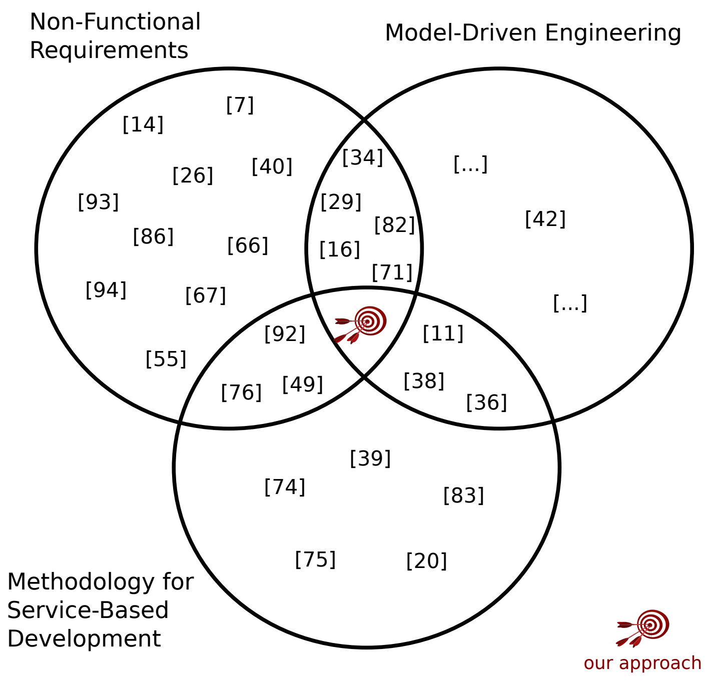


Figure 3: Related Works Analysis.

Figure 3 presents an overview of related work and their specific areas. The analysis helped us identify a gap for improving the development of service-based applications. Our approach uses the characteristics of the 3 areas[4] for proposing a methodology for developing service-based applications considering non-functional aspects.

Our work addresses the development of reliable service-based applications that uses services and non-functional aspects.

In the following chapters we describe a service-based methodology that we propose and that focusses on non-functional aspects description and modeling. For this, we use model-based development (MDD) and the classification of non-functional requirements described in this chapter, as well as the meta-model with NFR notation.

[4](i) Non-Functional Requirements; (ii) Methodology for Service-Based Development; and (iii) Model-Driven Engineering

# 3 πSOD-M: A Methodology for Building Reliable Service Based Applications

"*There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.*"

C.A.R. Hoare

This chapter presents $\pi$SOD-M (*Policy-based Service Oriented Development Methodology*), a methodology for modeling and developing reliable web service based applications using a MDA approach. We propose guidelines for building service compositions that include non-functional requirements (NFRs). The methodology focuses on the development of service-based system, considering non-functional aspects of the software being developed. These aspects are implemented as policies for the application services. The design of this aspects integrates both functional and non-functional requirements.

This chapter is organized as follows. Section 3.1 introduces the concepts, general structure and motivations to define a specific methodology for policy-based web service development. We use a motivational example for illustrating the concepts and principle of $\pi$SOD-M. Sections 3.2, 3.3 and 3.4 describe the development models and transformations of our methodology. They also how to apply the methodology for building concrete applications. Section 3.5 concludes the chapter.

## 3.1 $\pi$SOD-M

$\pi$SOD-M is a MDA (Model Driven Architecture) based methodology. It provides a framework for building service compositions considering their non-functional requirements. $\pi$SOD-M extends the SOD-M [36] method by adding the concept of *Policy* [42, 55] for representing NFR associated to service-based applications. $\pi$SOD-M also proposes the generation of a set of models at different abstraction levels, as well as transformations between these models.

$\pi$SOD-M's models represent both the cross-cutting aspects of the application being modelled, as well as the constraints associated to services. The systems cross-cutting concerns affect functional concerns, such as availability; recovery; and persistence aspects. Constraints are restrictions that must be respected during the execution of the application, for example the fact that a service requires an authentication for executing system functions.

$\pi$SOD-M defines a service oriented approach providing a set of guidelines to build service based information systems (SIS) and proposes to use services as first-class objects for the whole system development process. $\pi$SOD-M extends the SOD-M models to include capabilities to model NFRs. The SOD-M models that are being extended are: *use case model, extended use case model, service process model* and *service composition model.* $\pi$SOD-M provides a conceptual structure to: (i) capture the system requirements and specification in high-level abstraction models (computation independent models, CIMs); (ii) obtain the PIMs from such models (specification documents). The platform independent models (PIMs) are designed to specify the system details; (iii) transform such models into platform specific models (PSMs) that bundles the specification of the system with the details of the targeted platform; and (iv) serialize such model into the working-code that implements the system.

### 3.1.1 General Overview

Figure 4 presents the SOD-M models in the context of MDA architecture, and defines which models are extended by our approach. The $\pi$-*UseCase* model describes services requirements, constraints and quality requirements. The $\pi$-*ServiceProcess* model defines the concept of service contract for representing the service input and output data restrictions, as well as function restrictions. In this model we propose the definition of service contracts. The $\pi$-*ServiceProcess* model groups the constraints described in the $\pi$-*UseCase* model into contracts that are associated with services. The $\pi$-*ServiceComposition* model provides the concept of *Policy* that groups contracts with similar non-functional requirements. For example, security restrictions, such as contracts for authentication, privacy data access or transactions are grouped into a security policy.

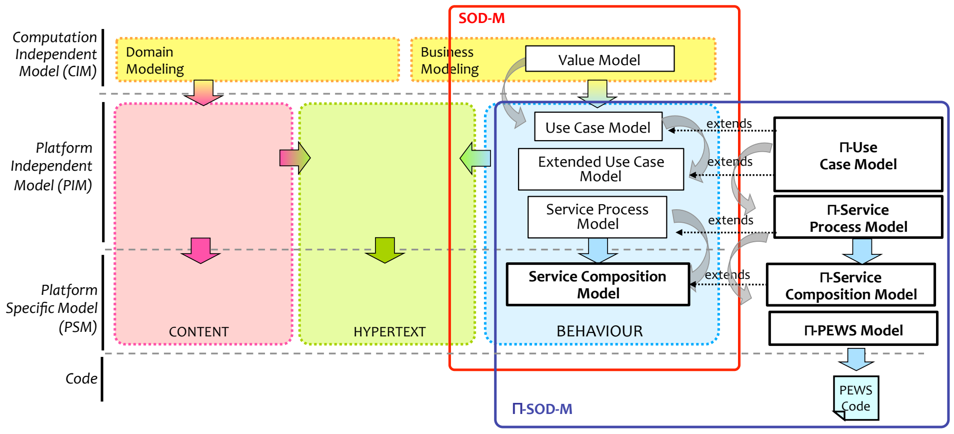


Figure 4: SOD-M and $\pi$SOD-M Extension Models.

In the $\pi$-*UseCase* model, use cases restrictions are designed as constraints. In the $\pi$-*ServiceProcess*, the constraints are grouped into service contracts, and use cases are refined to services or functions. Then, contracts are grouped into policies.

The $\pi$SOD-M methodology proposes the $\pi$-*PEWS* model as platform specific model. An instance of the $\pi$-*PEWS* model is generated from the $\pi$-*ServiceComposition* model, and it is a representation of a $\pi$-*PEWS* specification [27, 80]. From a $\pi$-*PEWS* model it is possible to generate a service composition code.

$\pi$SOD-M also defines model-to-model transformation rules (starting from the $\pi$-*UseCase* model) to $\pi$-*ServiceComposition* model; and uses model-to-text transformations to generate the corresponding implementation code in the $\pi$-*PEWS* language.

### 3.1.2 Development Process

$\pi$SOD-M uses the concept of MDA viewpoint, used as a technique of abstraction to focus on a particular aspect of the issue or proposed system. MDA defines, specifically, three viewpoints (figure 5):

- **Computation Independent Viewpoint:** This level focusses the environment of the system, as well as on its business and requirements specifications. At this moment of the development, the structure and system processing details are still unknown or undetermined. In $\pi$SOD-M, this level is represented as a list of business services from a requirements and business specification document.

- **Platform Independent Viewpoint:** This level focusses the system functionality, hiding the details of any particular platform. This specification defines those parts of the system that do not change from one platform to another. In πSOD-M, this level is modelled by the system use case, service process and service composition models.

- **Platform Specific Viewpoint:** This level focusses the functionality, without hiding the details of a particular platform, combining the platform independent view with the specific aspects of the platform to implement the system. In πSOD-M, the result of this level is the πPEWS specification [80] which represents as a platform specific model.

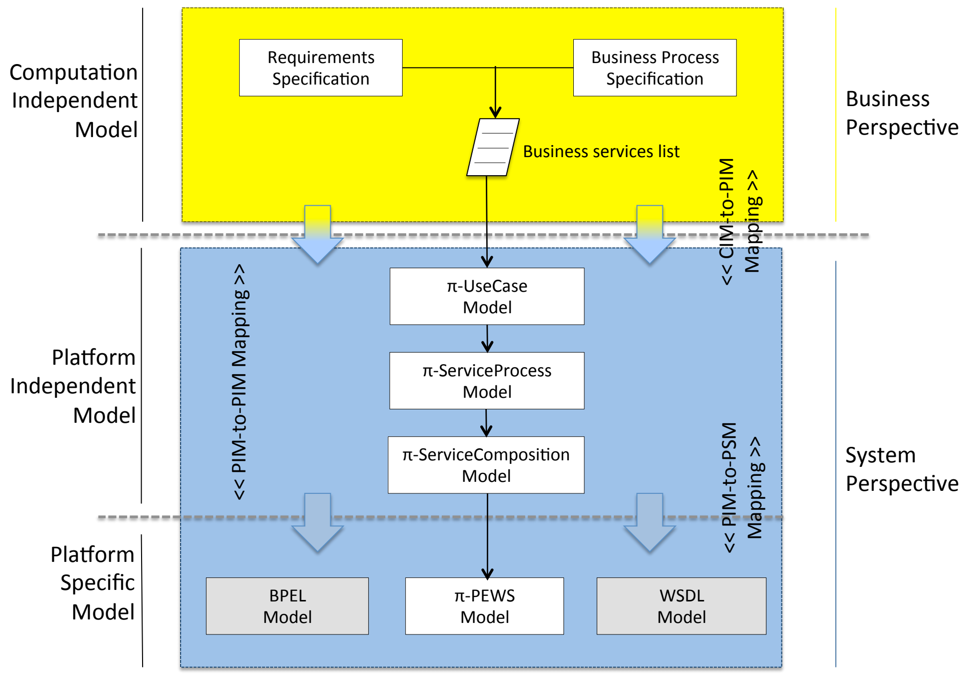


Figure 5: πSOD-M Development Process and Models.

Computation Independent Models (CIM) aim to represent the business view, while Platform Independent Models (PIM) and Platform Specific Models (PSM) aim to represent the information system view and detail the information system to be implemented to fulfill the requirements of a business environment.

Figure 5 presents the πSOD-M development process, which defines a service oriented approach providing the guidelines for building service-based information systems (SIS), that was the result of the SOD-M extension described in figure 4.

Next section presents the concepts that lead to modeling applications in πSOD-M. These concepts form the basis of our methodology for modeling the reliable service-based applications.

### 3.1.3 Methodology Concepts

The concepts presented in figure 6 represent the main elements of the methodology used for system application modeling. The $\pi$SOD-M's meta-model[1] concepts are used in all methodology meta-models. They describe the key concepts that must be modelled in a service-oriented application. The $\pi$SOD-M meta-model consists of the set of concepts for modelling and development applications that use MDA. These concepts represent the reliable service-based system development, and are present in the modelling any application.

Notice that the three $\pi$SOD-M methodology views (*Business, System* and *Policy*) are different to those levels proposed by the traditional MDA literature: *CIM, PIM* and *PSM*. In the $\pi$SOD-M meta-model, at different MDA levels, there are elements from the $\pi$SOD-M concepts.

All the proposed concepts and models for business modeling (CIM level), and for information system and policy modeling at PIM and PSM levels comprise the backbone for the $\pi$SOD-M based applications design. These concepts are important to identify the system requirements and properties throughout the development. Thus, the purpose of this section is to present a description of each concept, so that when applied and used, it can be seen in their proper modeling context.

The methodology relies on a set of concepts (represented as a meta-model) for modeling reliable applications. These concepts are structured in three views: *Business, System* and *Policy views* (figure 6):

- **Business view:** focuses on the business features that are the basis of an information system. The concepts that correspond to the *Business View* describe business elements. There are some concepts which correspond to both views (*Business and System Views*) and will be analyzed from both perspectives.
- **System View:** focuses on the main features and processes for the information system development. The *System View* concepts are used to describe the functionality and processing system.
- **Policy View:** focuses on non-functional requirements and business constraints of the information system. The concepts that correspond to the *Policy View* describe the NFRs and constraints related to the system functionalities. $\pi$SOD-M's main goal is to model

[1]*Meta-model* is the construction of a collection of "concepts" (things, terms, etc.) within a certain domain. A model is an abstraction of phenomena in the real world; a meta-model is yet another abstraction, highlighting properties of the model itself. A model conforms to its meta-model in the way that a computer program conforms to the grammar of the programming language in which it is written.

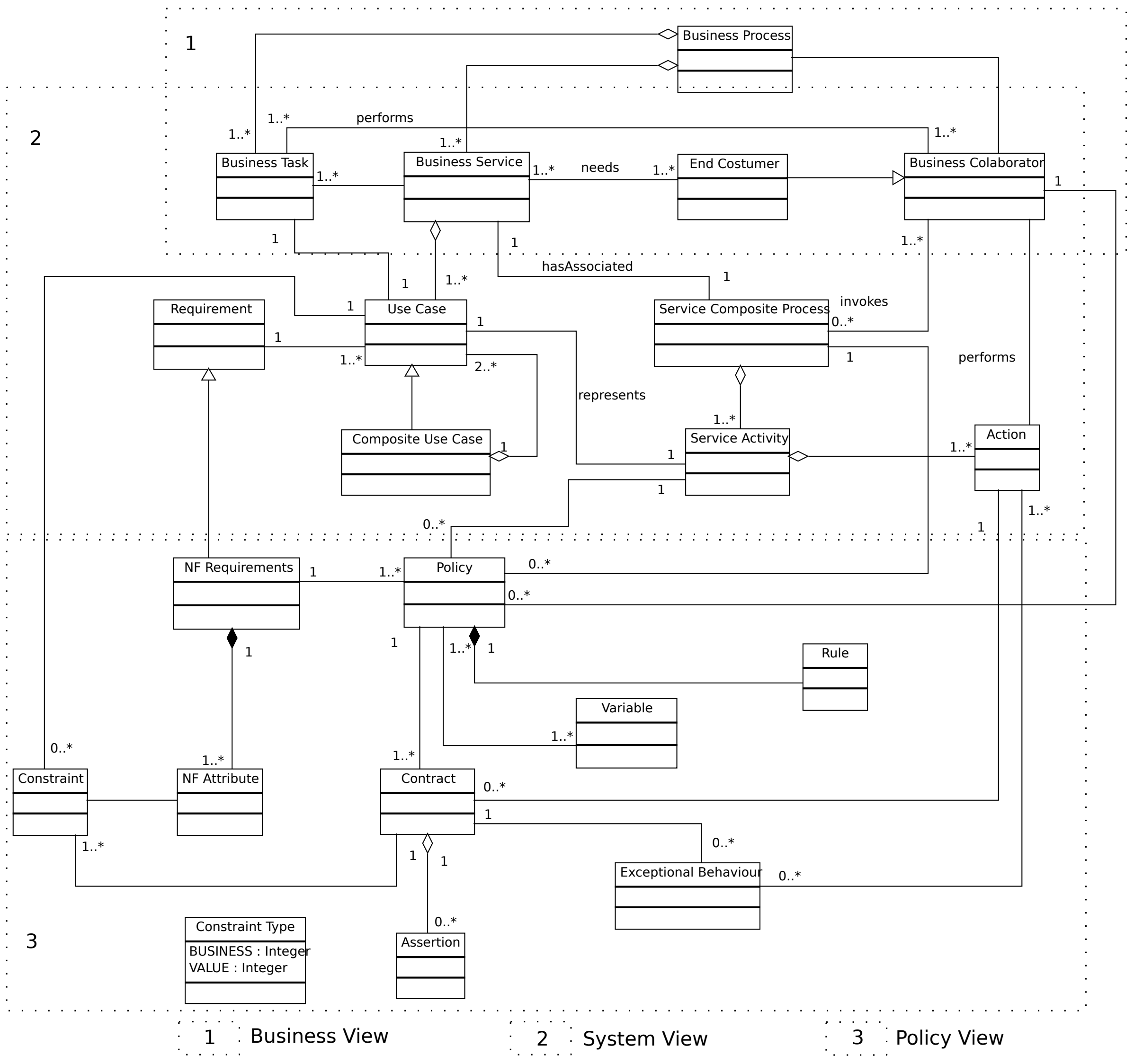


Figure 6: $\pi$SOD-M Concepts.

non-functional requirements considering the models proposed by the SOD-M methodology.

The $\pi$SOD-M concepts[2] are:

- **Business view:**
  - BUSINESS PROCESS - Represents the aggregation of logically related tasks that are carried out to achieve a given business result.
  - END CONSUMER* - Represents an entity that needs and consumes business services. End consumers are those who pay (either with money or using any other kind of value) to obtain a service that they use themselves.

[2]The concepts marked with * are also part of the **System view**

  - BUSINESS SERVICE* - Represents a result of a business process (or part of it) providing value for an end consumer.
  - `Business task*` – Represents a business function performed by a business collaborator as a part of a business process.
  - BUSINESS COLLABORATOR* - Represents an entity that collaborates in the business processes of an organization, performing certain tasks needed to provide a business service.

- **System view:**
  - REQUIREMENT - Represents a super type for functional and non-functional requirements. Thus, the use cases can be related to both types of requirements.
  - USE CASE - Represents a set of actions performed by the system to carry out part of a business service.
  - COMPOSITE USE CASE - Represents a set of actions performed by the system to carry out part of a business service, which can be broken down into different use cases, which may in turn be basic or composite.
  - SERVICE COMPOSITE PROCESS - Represents a set of logically related activities necessary for carrying out a business service.
  - SERVICE ACTIVITY - Represents a behaviour (set of individual actions) forming part of the execution flow of a business service.
  - ACTION - Represents a fundamental behaviour unit that is part of a service activity and describes some transformation or processing in the system being modelled.

- **Policy view:**
  - NON-FUNCTIONAL (NF) ATTRIBUTE - An attribute that describes the quality or characteristics of a functional requirement. For example *confidentiality* and *privacy* is an example for a non-functional attribute for the functional requirement *user registration*.
  - NON-FUNCTIONAL (NF) REQUIREMENT - A group of semantically correlated non-functional attributes (NFA). For example, *security* is an NF Requirement that comprises attributes such as *confidentiality* and *integrity*.
  - CONSTRAINT - A constraint prevents the system from achieving more than its goal. With the definition of constraints, the system can be more robust, and unexpected

problems can be solved before they happen. For example, in a banking system, the customer can only withdraw money if they have a positive balance in the account.

– CONSTRAINT TYPE - Represents the types of constraints, that may be on a system function or values. This entity can have their property setted as a business and data (*value) constraint (expressed as an attribute).

– CONTRACT - Is the formalization of obligations (requires) and benefits (ensures) of a function, service activity or component. The following questions can be used to define contracts: *What does it expect? What does it guarantee?* Contracts are crucial to software correctness and therefore they should be part of the design process. An interface is a kind of contract.

– ASSERTION - Represents a predicate or a state of the application before it runs (its preconditions), or the state when it is finished running (post-conditions);

– EXCEPTIONAL BEHAVIOUR - Are alternative execution paths if any condition or restriction is not respected. For example, if a user's password is not correct after three attempts, the user account is locked for security reasons.

– POLICY - A policy is a set of rules applied to a particular scope. This scope can be defined as an action, an activity, a function or a workflow. A policy is a composition of contracts applied to a non-functional application requirement. For example, a security policy of a system constraint may include authentication, access and data privacy properties.

– RULE - Represents information about an event, condition, and action where the event part represents the moment in which a constraint can be evaluated according to a condition represented by the condition part and the action to be executed for reinforcing it represented by the action part.

– VARIABLE - Represents a symbolic name given to some known information. A variable are related with a Policy. A Policy can have one or many variables. Each Variable has a name and a type.

The following next sections will describe the example scenario to better explain the $\pi$SOD-M meta-model and the methodology concepts.

### 3.1.4 Case Study

We use a case study in order to better describe specific details of the methodology proposed in this work.

Consider the following scenario: An organization wants to provide the service-based application "*To Publish Music*" that monitors the music a person is listening during some periods of time and sends the song title to this person's Twitter or Facebook accounts. More specifically, this social network user will have their status synchronized in both Twitter[3] and Facebook[4], with the title of the music the user is listening in Spotify[5]. The user may also want to download music. For this, the user needs to process the purchase of music they want to download. Payments will be processed via PayPal or credit card. All services are available via Spotify and our application needs to interact with Spotify users, so they can listen music, publish on Facebook and Twitter, and purchase music online. The corresponding use case model is illustrated in in figure 7.

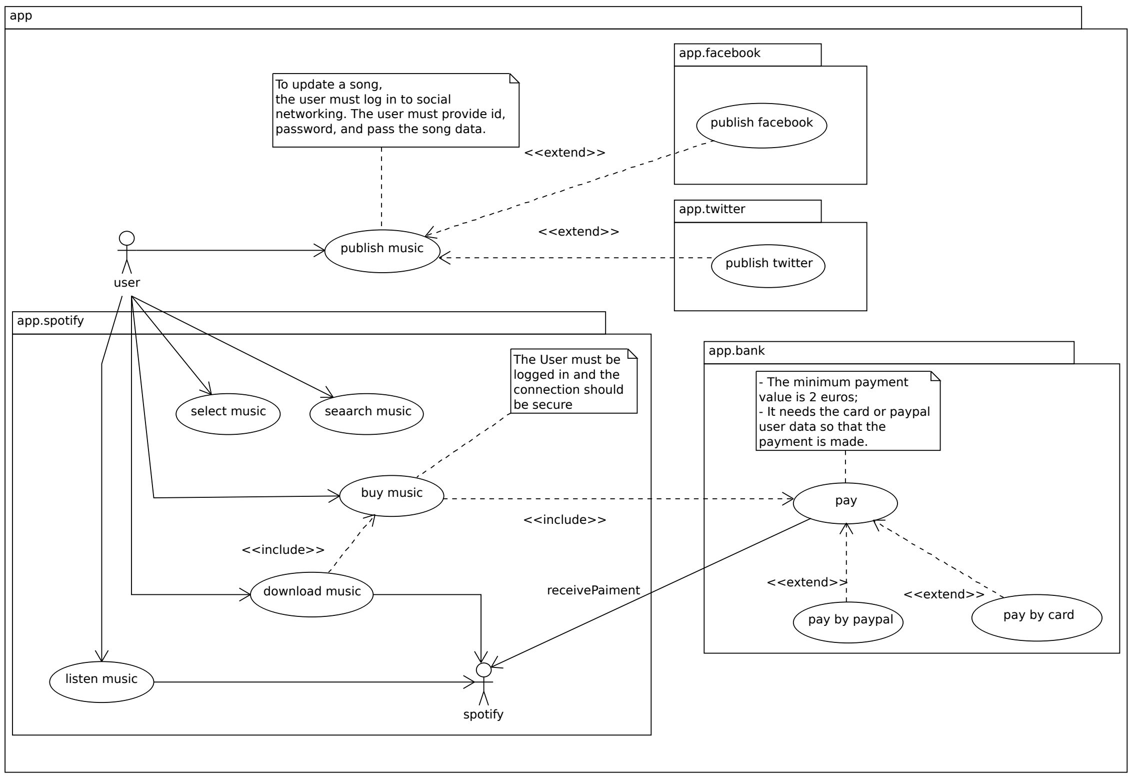

app
app.facebook
publish facebook
To update a song, the user must log in to social networking. The user must provide id, password, and pass the song data.
<<extend>>
app.twitter
<<extend>>
publish music
publish twitter
user
app.spotify
app.bank
The User must be logged in and the connection should be secure
select music
seaarch music
- The minimum payment value is 2 euros;
- It needs the card or paypal user data so that the payment is made.
buy music
pay
<<include>>
<<include>>
download music
receivePaiment
<<extend>>
<<extend>>
pay by paypal
pay by card
listen music
spotify

Figure 7: Use Case Model "*To Publish Music*" - Spotify Music Service.

Figure 8 shows the activity model for our scenario. It starts by contacting the music service Spotify for retrieving the user's musical status. If the user wishes to download a particular song, they can download the song to purchase (activities *Buy Music* and *Download Music*). Finally

[3]Twitter is an online social networking service and micro-blogging service that enables its users to send and read text-based posts of up to 140 characters, known as "*tweets*".

[4]Facebook is a social networking service and website. Users may join common-interest user groups, organized by workplace, school or college, or other characteristics, and categorize their friends into lists

[5]Spotify is a music streaming service offering digitally restricted streaming of selected music from a range of major and independent record labels. Spotify allows registered users to integrate their account with existing Facebook and Twitter accounts.

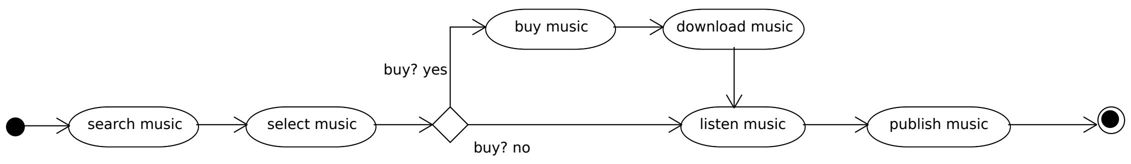


Figure 8: Service Process Model "*To Publish Music*"

the user can publish the music. The activity *publish music* can also be modelled as a parallel workflow for updating different social networks, as follows: Twitter and Facebook services can be contacted in parallel for updating the user's status with the corresponding song title.

The execution flow is described as follows. Each action from the activity diagram represents one or more functions in the application. Each function or service has an interface, consisting of its input and output, and the specification of how to call this function/service. An activity must be executed according to the rules identified in the use case model. For example, the "*publish music*" action (figure 8), can be refined into two flows: (i) a flow to publish on Twitter and (ii) another to update the Facebook, since each service has different rules for update and login.

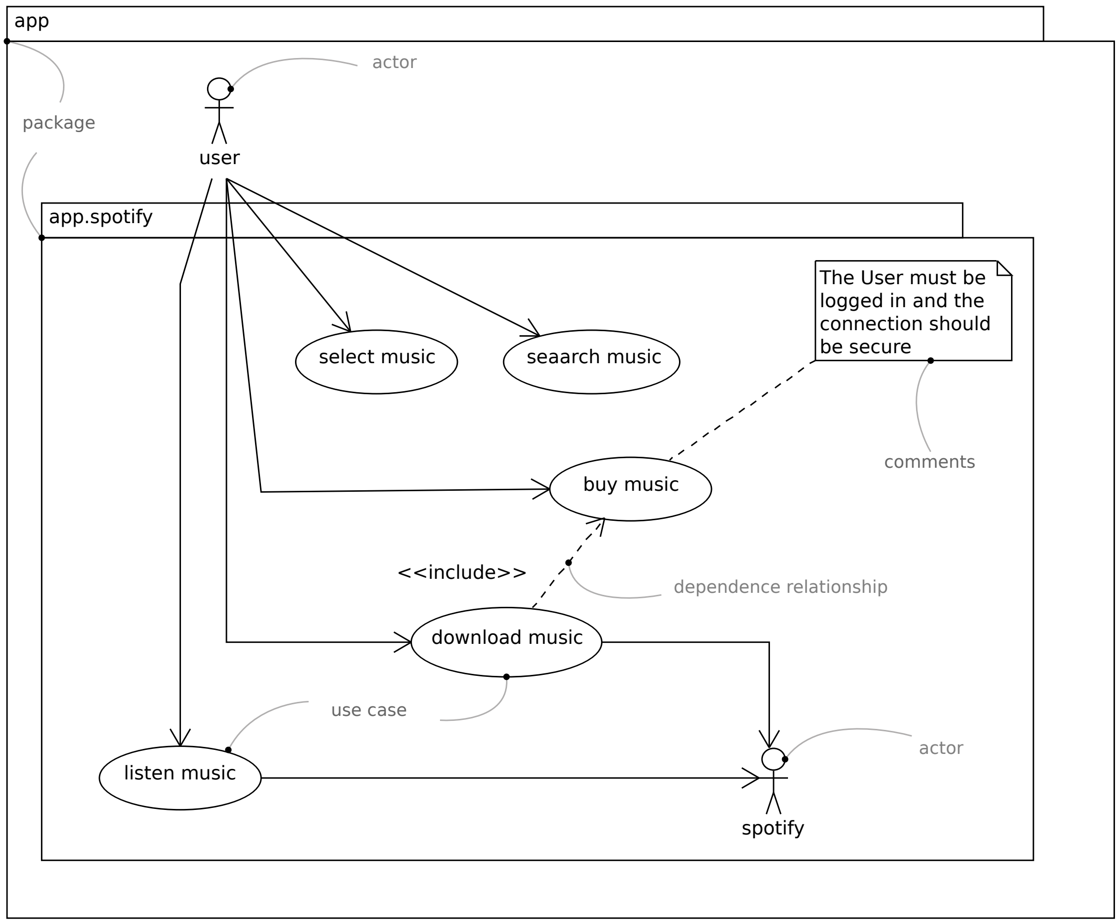
Given a set of services with their exported methods (known in advance or provided by a service provider), building service-based applications is a task that implies expressing an application logic as a service composition together with some restrictions.

Both models presented in figures 7 and 8 use the traditional UML to design requirements in use case diagram and workflow actions in the activity diagram. Figure 9 presents some of the main UML concepts that are used to model system diagrams. The structure of each $\pi$SOD-M model will be detailed in accordance with their respective meta-model and transformations necessary for achieving the aim of modeling reliable applications.

## 3.2 Platform Independent Models

The *Platform Independent Models (PIM)* are used for modeling the system functionality, abstracting technological details.

The models proposed at $\pi$SOD-M's PIM level are (figure 5): $\pi$-*UseCase,* $\pi$-*ServiceProcess* and $\pi$-*ServiceComposition*. They provide concepts for expressing the design of: (*i*) system requirements and constraints; (*ii*) external services and their compositions; (*iii*) execution flow; (*iv*) input and output service restrictions; (*v*) service contracts and actions; and (*vi*) policies for the implementation of business services. We use these models to provide a representation of the system and to specify its restrictions.



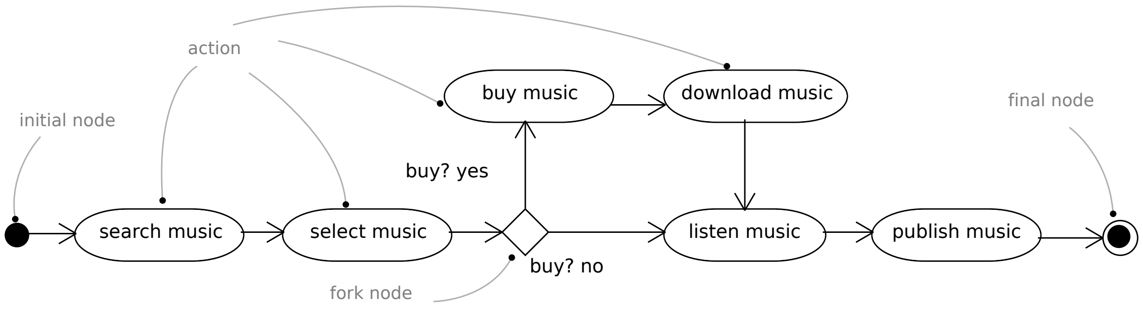


Figure 9: Traditional UML Concepts.

### 3.2.1 *π-UseCase* Model

The *π-UseCase* (figure 14) is used to describe the functionalities (services and other functionalities) of the system. The services are represented in this model as use cases. We design this model by using the UML use cases, with a small extension for considering constraints. Application constraints are represented by stereotyped use cases and comments related to them. Each constraint describes what will be checked for a particular application function be executed.

Figure 10 shows[6] an example of how to describe constraints on use cases in $\pi$SOD-M. We use part of our case study to present them. Each use case may have multiple constraints. In this example the *publish music* use case has an authentication constraint (*user must authenticate*). For a song to be published on any social network it is necessary to provide data privacy. The constraint type concerns the authentication values and the constraint has an non-functional attributes. The use case is also related with requirement, a business service and a non-functional requirement properties.

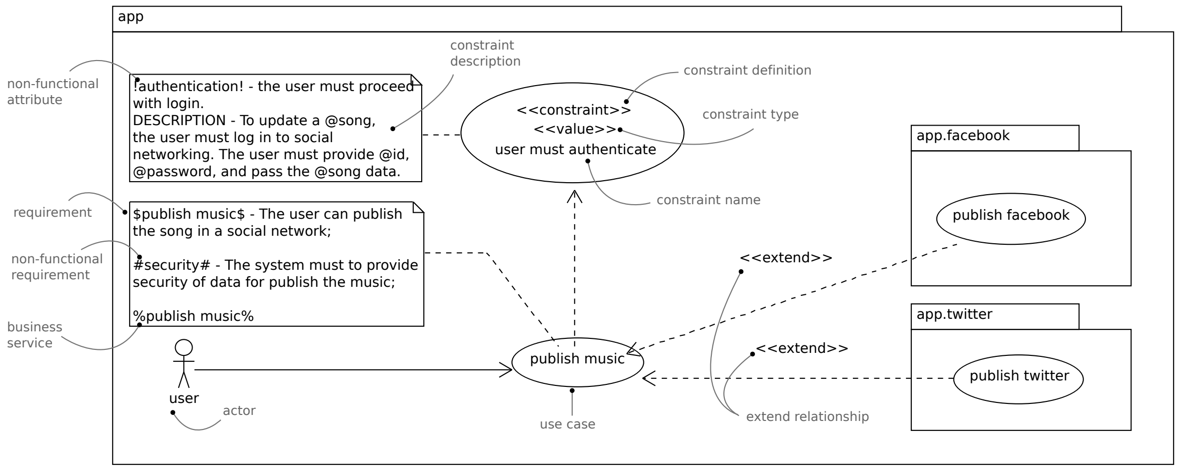

app
constraint description
constraint definition
non-functional attribute
!authentication! - the user must proceed with login.
DESCRIPTION - To update a @song, the user must log in to social networking. The user must provide @id, @password, and pass the @song data.
<<constraint>>
<<value>>
user must authenticate
constraint type
app.facebook
constraint name
requirement
$publish music$ - The user can publish the song in a social network;
non-functional requirement
#security# - The system must to provide security of data for publish the music;
%publish music%
business service
publish facebook
<<extend>>
app.twitter
<<extend>>
publish music
publish twitter
user
actor
use case
extend relationship

Figure 10: $\pi$-UseCase model Example.

### 3.2.1.1 $\pi$-*UseCase* Diagram, Terms and Concepts

The $\pi$-*UseCase* model describes the functional aspects of the application being developed and the restrictions that applies to each of them. These restrictions are modeled using the concepts: (i) constraint type and (ii) non-functional requirement with its specific attribute.

A $\pi$-*UseCase* model is a way to represent features of an application, services, constraints, and consumers of application functionality.

All constraints are described by stereotyped use cases. Every relationship between an use case and a constraint is done through a dependency relationship (from UML). All use cases that are related with constraint must be represented such as described in figure 11. A constraint is defined by a stereotype, containing its *type* and *name*. The CONSTRAINTS are represented by a stereotyped use case (*«constraint»*).

Each use case is related with one or more system requirements. Moreover, these use cases are related to non-functional requirements necessary for the proper execution of the application

[6]The annotation in this figure are not part of the model. They are included for didactic explanation only. The same applies for the other figures in this section which present the same type of annotation.

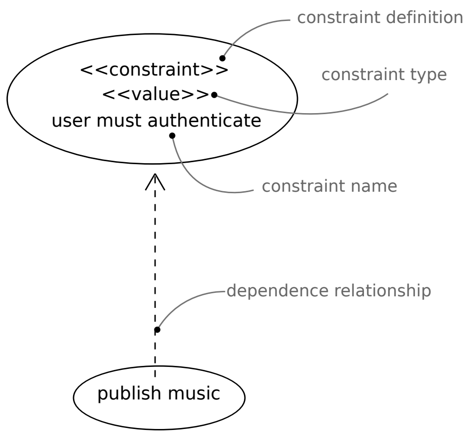


Figure 11: Constraint Representation.

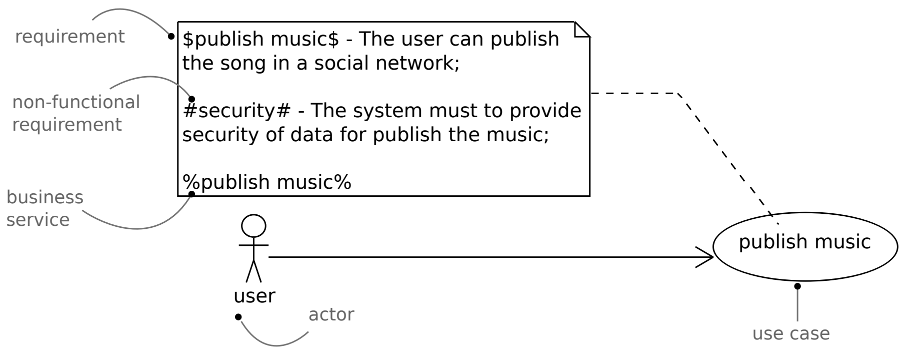

requirement
$publish music$ - The user can publish the song in a social network;
non-functional requirement
#security# - The system must to provide security of data for publish the music;
business service
%publish music%
publish music
user
actor
use case

Figure 12: Non-functional Requirement Representation.

function. Similarly, the function described by an case of use can be part of a business service (in the case of service remote calls). These information to a specific use case may be described through comments as described in figure 12. All these information must be associated with the use case and its constraints.

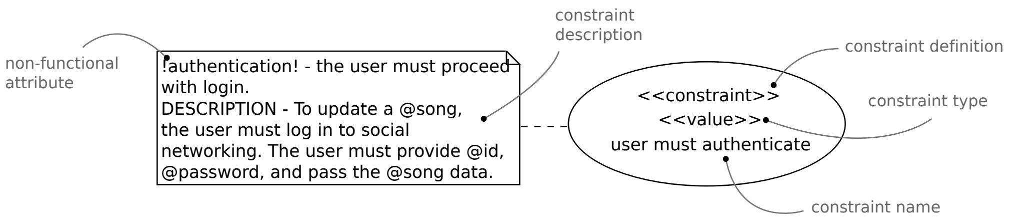

non-functional attribute
!authentication! - the user must proceed with login.
DESCRIPTION - To update a @song, the user must log in to social networking. The user must provide @id, @password, and pass the @song data.
constraint description
constraint definition
<<constraint>>
<<value>>
user must authenticate
constraint type
constraint name

Figure 13: Non-functional Attribute Representation.

The constraints are always related to non-functional requirements. More specifically, each constraint describes a non-functional attribute, *e.g. authentication* (presented in figure 13). A constraint consists of the constraint description and the identification of variables be verified (by an @, for example @*song*), the identification of the non-functional attribute and its description.

These concepts that are described in the $\pi$SOD-M meta-model and used in $\pi$-*UseCase* represent the concepts we identified necessary for reliable service-based modeling.

#### 3.2.1.2 Meta-model

The concepts modelled in the $\pi$-*UseCase* meta-model are: BUSINESS SERVICE, END CONSUMER, REQUIREMENT, USE CASE, COMPOSITE USE CASE, NON-FUNCTIONAL REQUIREMENT, NON-FUNCTIONAL ATTRIBUTE and CONSTRAINT. Figure 14 presents the $\pi$-*UseCase* meta-model and the relationship between each concept. We highlight the elements that represent the *policy view*.

In the $\pi$-*UseCase* an END CONSUMER is represented by an ACTOR (figure 14). An ACTOR is related to USE CASES (from the original UML definition), while a COMPOSITE USE CASE is a set of actions performed by the system which can be broken into different USE CASES.

BUSINESS SERVICE aggregates several USE CASES (figure 14), simply put, a service can be expressed by one or more use cases. Notice that not always an use case will be part of a business service. However, using our approach, we can identify in the early stages of modeling, which use cases are related to business services.

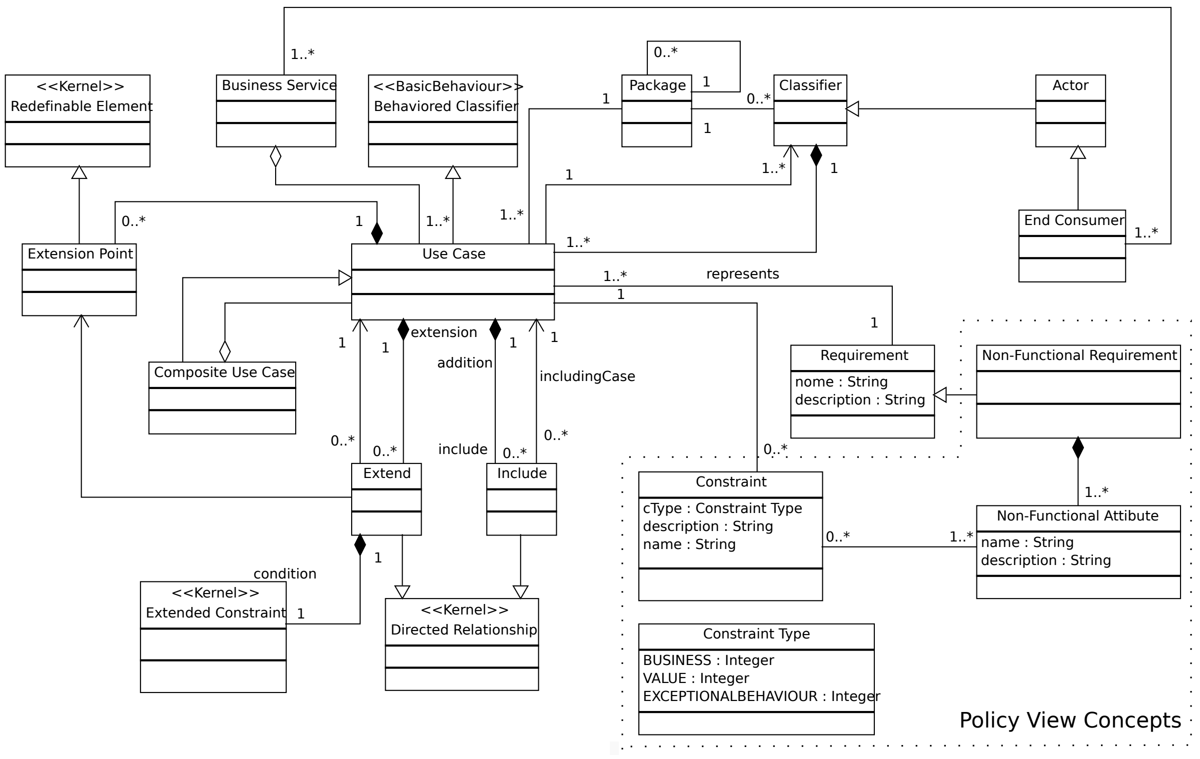


Figure 14: $\pi$-UseCase Concepts (Meta-model).

The BUSINESS COLLABORATORS concept are represented in the $\pi$-*UseCase* model through PACKAGES. A BUSINESS COLLABORATOR represents an external service or a system that interact with the application that are being modelled. Each BUSINESS COLLABORATORS combines the features described in each PACKAGE. Thus, the services functions can be specified being grouped into packages (figure 15).

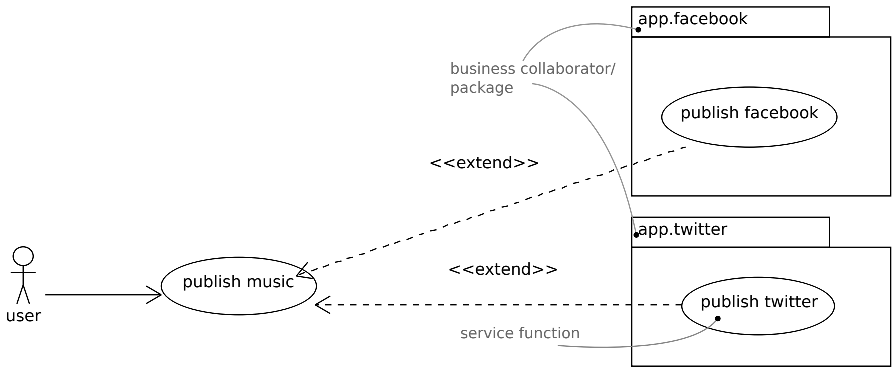


Figure 15: Business Collaborator / Package Representation.

The NON-FUNCTIONAL REQUIREMENT and NON-FUNCTIONAL ATTRIBUTE concepts are represented as part of description of USE CASES and CONSTRAINTS. For model this concepts the designer will use special tags to represent them (figures 12 ans 13).

An USE CASE may have several CONSTRAINTS. Each CONSTRAINT has its name, description, and which ones should be checked during the execution of the application. Each CONSTRAINT is represented as a stereotyped («`constraint`») use case. A restriction may be a restriction on data (VALUE CONSTRAINT), represented as the stereotype «`value`» (figure 11); or on business rules (*Business Constraint*), represented as the stereotype «`business`»; and EXCEPTIONAL BEHAVIOUR constraints, which represents constraint violation for the use case execution, represented as the stereotype «`exceptional_behaviour`».

The $\pi$-*UseCase* meta-model also represents the use of include and extend relations among the different use cases identified. The semantics are the same as in the traditional UML use case model. An include relation specifies the existence of a flow of events in which the base use case includes the behaviour of the other use case and an extend relation specifies that the behaviour of a base use case can be optionally extended by the behaviour of another use case.

### 3.2.1.3 UML Concepts Representation

Once described the meaning of each $\pi$-*UseCase* concept in the meta-model, it is important to know how specific models can be created from this general representation. Every $\pi$-*UseCase* model created to describe application features must follow the meta-model concepts. The $\pi$-*UseCase* meta-model describes what can be specified in each model.

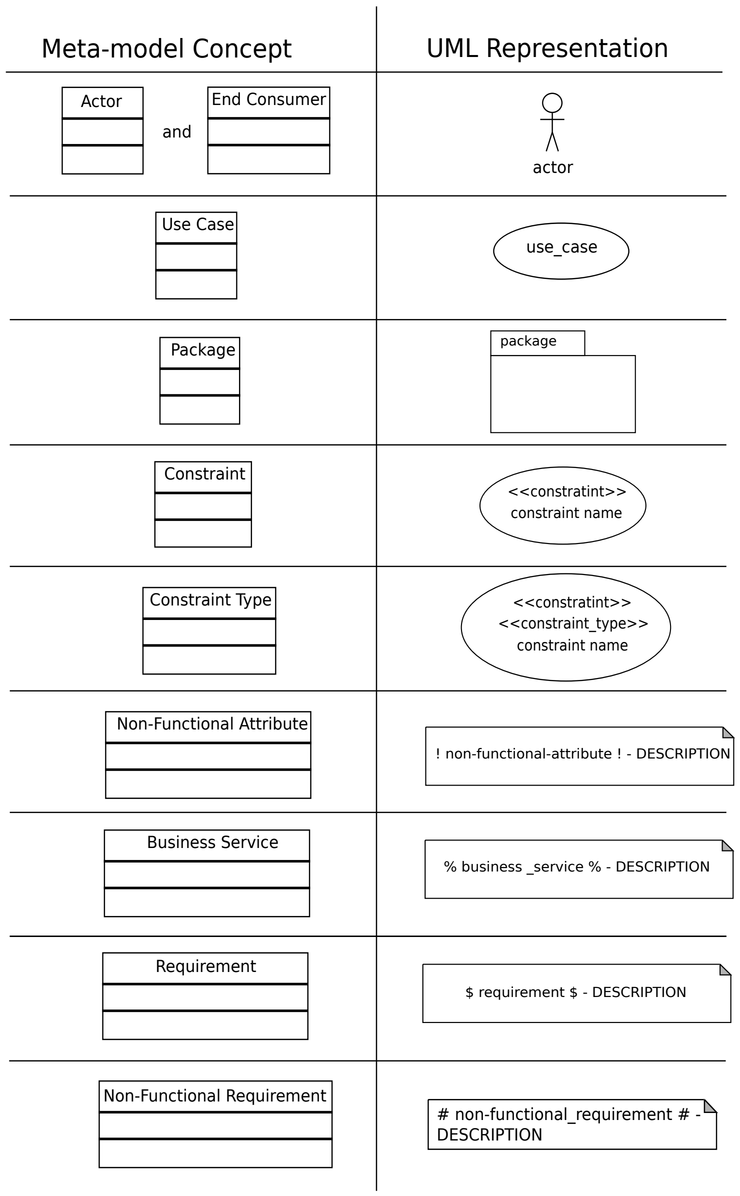


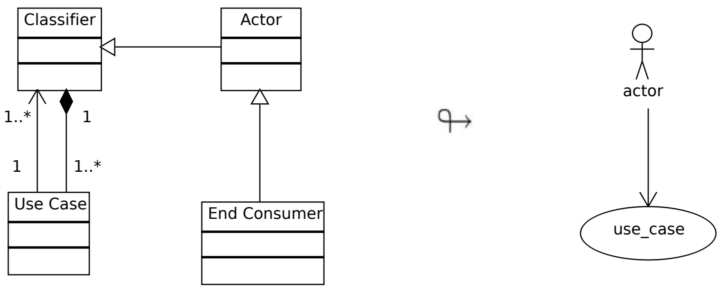
Figure 16: $\pi$-*UseCase* Concepts Represented as UML Elements.

The $\pi$-*UseCase* concepts are modelled as described in figure 16 and the relationship between these concepts is also applied to the UML elements. Figures 17 and 18 detail how to represent the relationship between the meta-model concepts in a $\pi$-*UseCase* model.

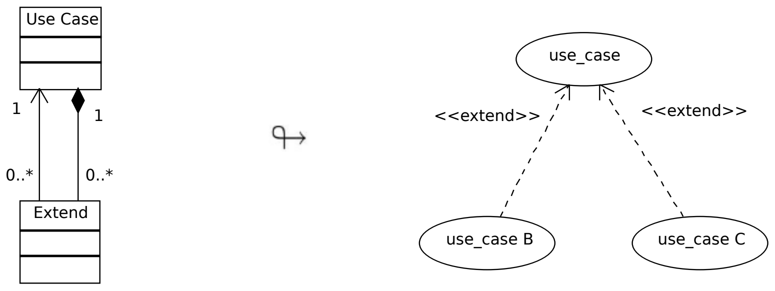
The *loop arrow* symbol ($\looparrowright$) is used in figures 17 and 18 to relate the concepts of the meta-model and the UML elements used in the representation of the concepts.

means how the $\pi$-*UseCase* meta-model concepts (left side) can be modelled as an UML model (right side).

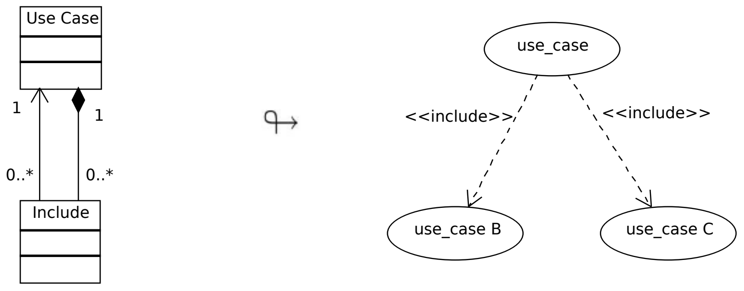
An ACTOR is an abstraction of the END COSTUMER concept, presented in $\pi$SOD-M meta-model. Thus, it is possible to associate several ACTORS with an USE CASE and many USE



(a) *Use Case* and *Actor* Model



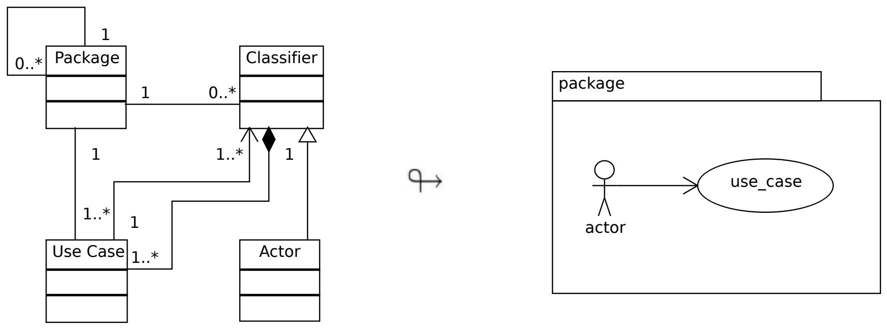
(b) *Extended Use Case* Model



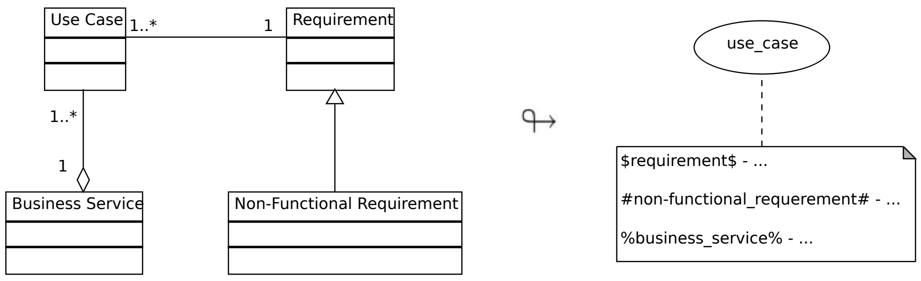
(c) *Include Use Case* Model

Figure 17: $\pi$-*UseCase* Model Representation

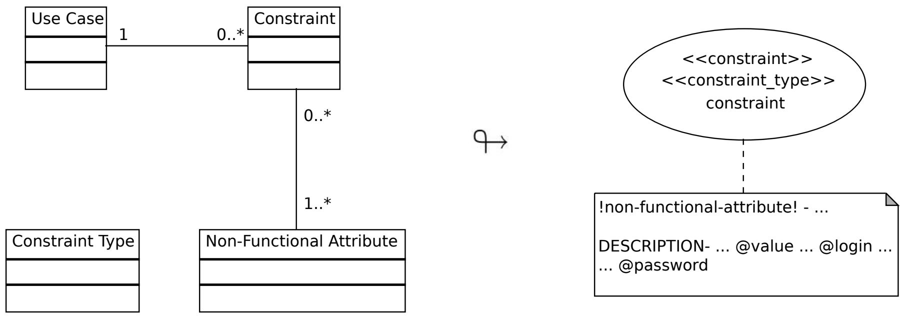
CASES with an ACTOR (END COSTUMER). The original use case meta-model defines that CLASSIFIER is a superclass of ACTOR. A CLASSIFIER is a category of UML elements that have some common features, such as *attributes* or *methods*. The relationship between these entities is many to many (figure 17a). An USE CASE can also be related with an EXTENDS or an INCLUDE entity in the $\pi$-*UseCase* model, and this model preserve the original UML standard semantic.



(a) *Package* Model



(b) *Use Case*, *Requirement* and *Non-Functional Requirement* Model



(c) *Constraint* Model

Figure 18: $\pi$-*UseCase* Model Representation (2)

EXTENDS and INCLUDE are types of stereotyped dependence relationships («`extend`» and «`include`» in figures 17b and 17c, respectively). The EXTENDS relationship can modify the behaviour of the base use case. Suppose someone wants to close a bank account, however still have some money there. Before closing the account, there should be a withdrawal or transfer of money in this account, so that the balance is zero. This process is modelled with a EXTENDS dependence between the *close account* use case and the *withdrawal money* and *transfer money* use cases. The INCLUDE relationship includes the steps from one use case into another.

A PACKAGE clause and its relationship with USE CASE and ACTOR can be modelled in *π-UseCase* model. The original semantics was preserved and it is possible have different levels of packages, due to PACKAGE entity auto reference, one to many. Each package can have many use cases and actors (18a).

The REQUIREMENTS, NON-FUNCTIONAL REQUIREMENTS and BUSINESS SERVICES must be associated with a specific USE CASE. At this level of modelling, these information are represented as comment *tags*, they are: *$ ... $* for REQUIREMENTS, *# ... #* for NON-FUNCTIONAL REQUIREMENTS, and *% ... %* for BUSINESS SERVICE. A BUSINESS SERVICE is a aggregation of USE CASE, and each USE CASE must be related with a system requirement REQUIREMENTS. Figure 18b presents the relationship of this concepts, and how represent this information in UML.

All USE CASES may also be associated with a set of CONSTRAINTS. A CONSTRAINT is a stereotyped use case that describes the CONSTRAINT TYPE and its restriction. Additional information is given by comment tags, *! ...!* for NON-FUNCTIONAL ATTRIBUTE, and @ ... @ for candidate variables or values that must be checked during the system execution, as presented in figure 18c. The relationship between USE CASES and CONSTRAINT is made by a non-stereotyped dependence arrow.

#### 3.2.1.4 *To Publish Music* Use Case

Considering the example scenario, the "*publish music*" use case can update the Facebook or Twitter user status. Therefore, it is necessary to perform a social network authentication with the user's data. Each social network uses different services and different forms of authentication. The "*authentication*" constraint is required to update a music status. The restriction is stereotyped as a «`value`» constraint, because the *user's id* and *password* are verified. The NON-FUNCTIONAL REQUIREMENT that is being modelled for this use case is *Security*, because, considering table 4, "*authentication*" is a NON-FUNCTIONAL ATTRIBUTE of *Security*. These information will be refined and detailed in the following phases.

Figure 19 shows the model represented in figure 7 for the *buy music, download music, listen music* and *pay* use cases. The model was designed considering the meta-model representation detailed in figures 17 and 18, and also the *π-UseCase* meta-model. Notice that from *π-UseCase* concepts is possible to describe the actors, use cases, constraints, non-functional requirements and their attributes. For example, the *pay* use case (figure 19), there are extended use cases, constraints, packages, non-functional requirements and non-functional attributes related to this function. The modeling was based on the described representation of the *π-UseCase* meta-model. The extended use case representation presented in figure 17b is used in the model to expressed the *pay by PayPal* and *pay by card* extended use cases. The include use case representation presented in figure 17c is used in the model to express the included relation of *buy music* and *pay* use cases. The package representation presented in figure 18a is expressed in the example by the *app.bank* package and the *pay* use case relation. The representation described in figures 18b and 18c is expressed by the *pay* use case properties' description. The combination of a *π-UseCase* meta-model's entities means that all entities presented in figures 10 and 19 can be modelled to express the quality requirements.

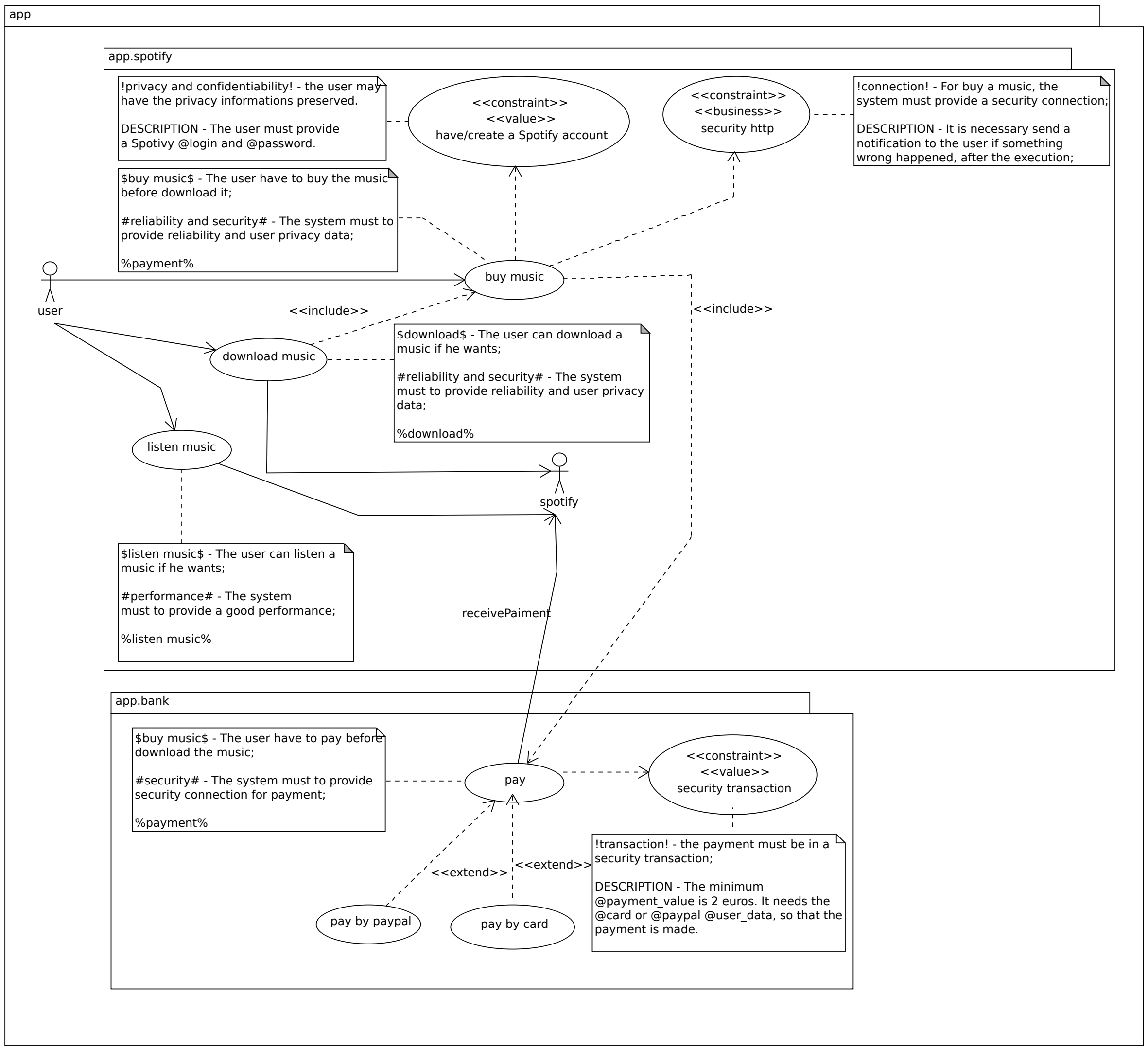
The use cases that have restrictions are "*buy music*" and "*pay*". The process of buying a song requires the user's private data for a Spotify account, and also a secure connection, represented as «`value`» and «`business`» constraint, respectively. For payment, the user must provide the data of payment card or PayPal account login and password, represented as «`value`» stereotype . Other restriction is that the minimum payment value is 2 euros. The NON-FUNCTIONAL REQUIREMENT that are being modelled for this use cases are *Security* and *Reliability*. The specific NON-FUNCTIONAL ATTRIBUTES for this restriction is *Transaction* and *Confidentiality* (see figure 19).

After modeling the system services and features, and its restrictions, it is necessary to model the services process and its execution restrictions. For this, the methodology uses a service process diagram to describe the services execution and contracts. Next we present the *π-ServiceProcess* model. This model refines the application constraints through the workflow model.

### 3.2.2 *π-ServiceProcess* Model

A *Service Process* is a collection of related, structured activities that produce a specific service or product for a particular customer. It can be visualized with a flowchart as a sequence of activities with interleaving decision points. A set of linked activities that take an input and transform it to create an output. Ideally, the transformation that occurs in the process should add value to the input and create an output that is more useful and effective. To represent these properties our methodology proposes the *π-ServiceProcess* model.

The non-functional requirements are specified in this model using the concepts of contracts



!privacy and confidentiability! - the user may have the privacy informations preserved.

DESCRIPTION - The user must provide a Spotivy @login and @password.

!connection! - For buy a music, the system must provide a security connection;

DESCRIPTION - It is necessary send a notification to the user if something wrong happened, after the execution;

$buy music$ - The user have to buy the music before download it;

#reliability and security# - The system must to provide reliability and user privacy data;

%payment%

$download$ - The user can download a music if he wants;

#reliability and security# - The system must to provide reliability and user privacy data;

%download%

$listen music$ - The user can listen a music if he wants;

#performance# - The system must to provide a good performance;

%listen music%

$buy music$ - The user have to pay before download the music;

#security# - The system must to provide security connection for payment;

%payment%

!transaction! - the payment must be in a security transaction;

DESCRIPTION - The minimum @payment_value is 2 euros. It needs the @card or @paypal @user_data, so that the payment is made.

Figure 19: Use Case Model With Constraint Definition - Scenario Example Refined.

and assertions over functions. These contracts may represent the control of privacy information, performance, reliability, and so on.

### 3.2.2.1 $\pi$-*ServiceProcess* Diagram, Terms and Concepts

Systems that use third party services add value to the application because the services are already available and validated. If the services do not offer any quality warranty, these quality requirements are the responsibility of the application designer. Thus, the contracts of services will be modeled as a wrapper which describes restrictions on input and output information on particular functions or a set of them.

$\pi$SOD-M proposes modeling *Service Process* diagram ($\pi$-*ServiceProcess* model), that is defined considering the functional and non-functional requirements designed in a $\pi$-*UseCase* model.

The $\pi$-*ServiceProcess* model represents the service processes and its contracts. It models the set of system activities and describe contracts for each activity or the process as a whole. Each service identified in the previous model ($\pi$-*UseCase*) is mapped to an action.

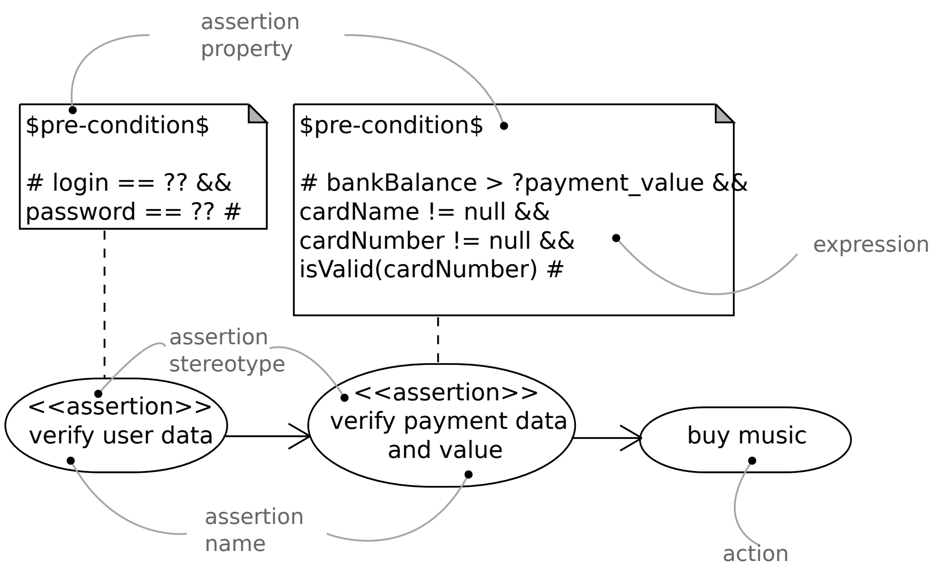


Figure 20: Assertion Representation.

A $\pi$-*ServiceProcess* model represents the system process, actions, assertions and contracts. This model describes the workflow for the modelled application behaviour. The behaviour is represented by each action and the restrictions by the assertion description. All assertions are described by stereotyped actions.

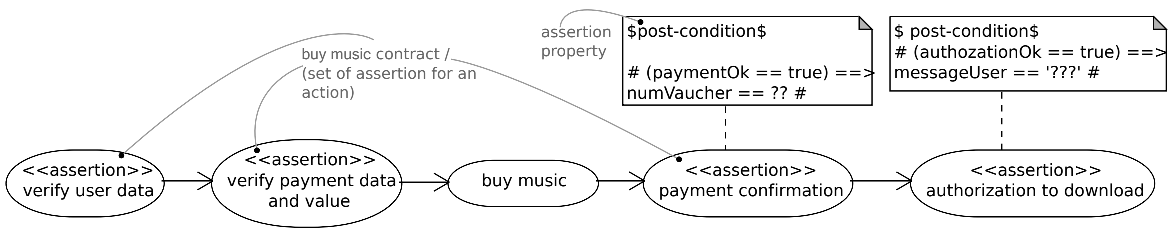


Figure 21: Contract Representation.

Figures 20 and 21 present the assertion and contract representation in this model. An Assertion is represented in UML, and a set of assertions forms a contract. Assertions describe restrictions on a single action, but actions may have different restrictions. Figure 20 shows how an assertion is modelled, with their property type, name and acceptable values (expressed through expressions), while figure 21 shows how this set of assertions form a single action contract.

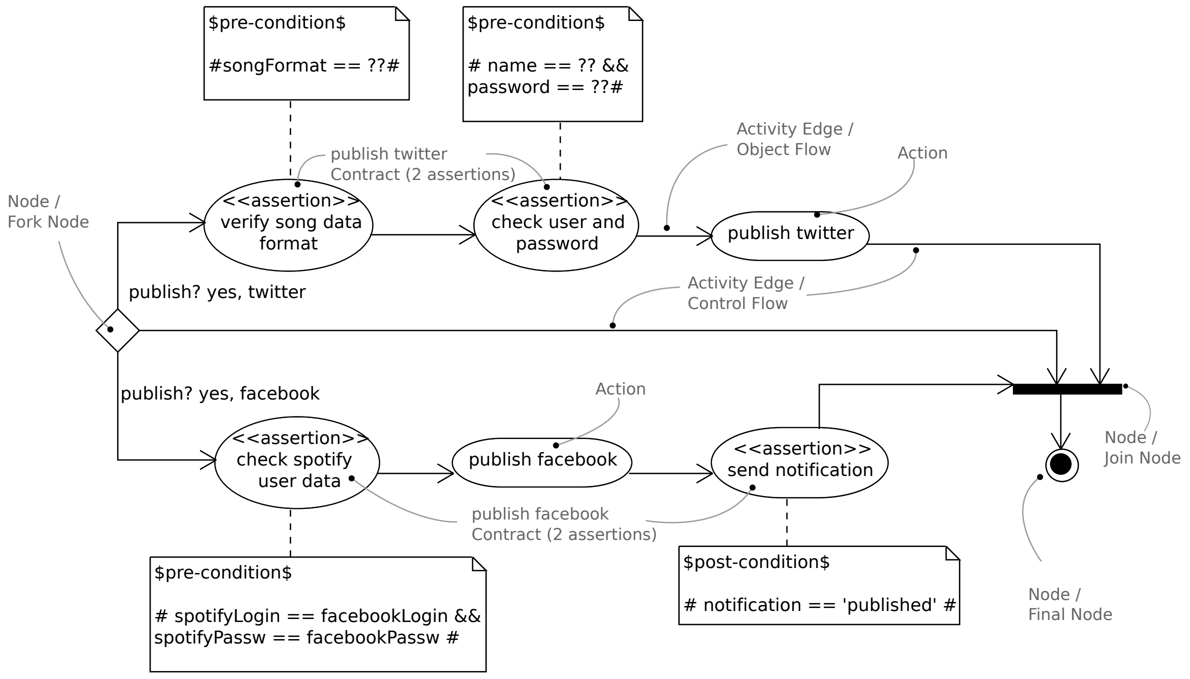

$pre-condition$
#songFormat == ??#
$pre-condition$
# name == ?? && password == ??#
Activity Edge / Object Flow
Action
publish twitter Contract (2 assertions)
Node / Fork Node
<<assertion>> verify song data format
<<assertion>> check user and password
publish twitter
publish? yes, twitter
Activity Edge / Control Flow
publish? yes, facebook
Action
<<assertion>> check spotify user data
publish facebook
<<assertion>> send notification
Node / Join Node
publish facebook Contract (2 assertions)
$pre-condition$
# spotifyLogin == facebookLogin && spotifyPassw == facebookPassw #
$post-condition$
# notification == 'published' #
Node / Final Node

Figure 22: Node, Activity Edge and Contract Representation.

Figure 22 shows the representation of the general concepts of this model, nodes, activity edges, actions and contracts. This figure also refines the information for publish a song in a social network.

#### 3.2.2.2 Meta-model

Considering the $\pi$SOD-M concepts (figure 6), the $\pi$-*ServiceProcess* meta-model (figure 23) describes the system functionalities to be modelled in a contract based service process. The $\pi$SOD-M concepts modelled in the $\pi$-*ServiceProcess* meta-model are: CONTRACT, ASSERTION, EXCEPTIONAL BEHAVIOUR, ACTIVITY, SERVICE ACTIVITY, ACTION and CONSTRAINT.

Restrictions on activities and services are not described in the original SOD-M proposal. The concepts of CONTRACT, ASSERTION and EXCEPTIONAL BEHAVIOUR make it possible to model restrictions on services execution, ensuring greater reliability application. Entities highlighted in $\pi$-*ServiceProcess* meta-model (figure 23) are the difference from the original SOD-M service process model.

A specific $\pi$-*ServiceProcess* model shows the set of logically related activities that need to be performed in a service-based system. So, the activities of this model represent a behaviour that is part of the system workflow. $\pi$SOD-M proposes representing this model using

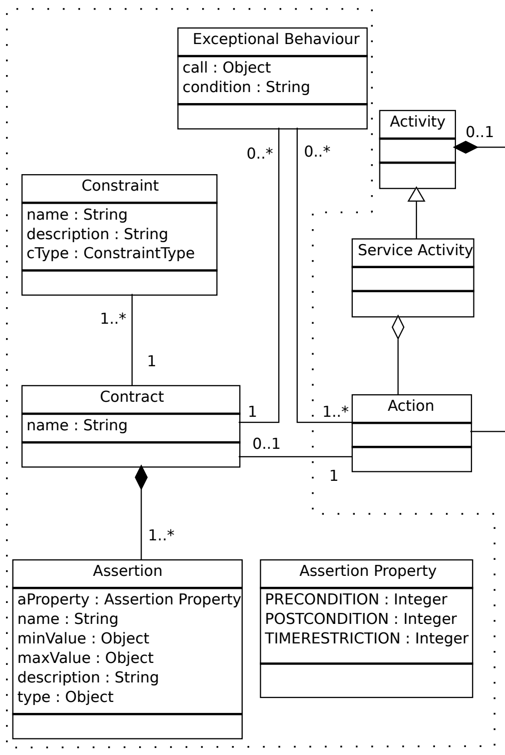
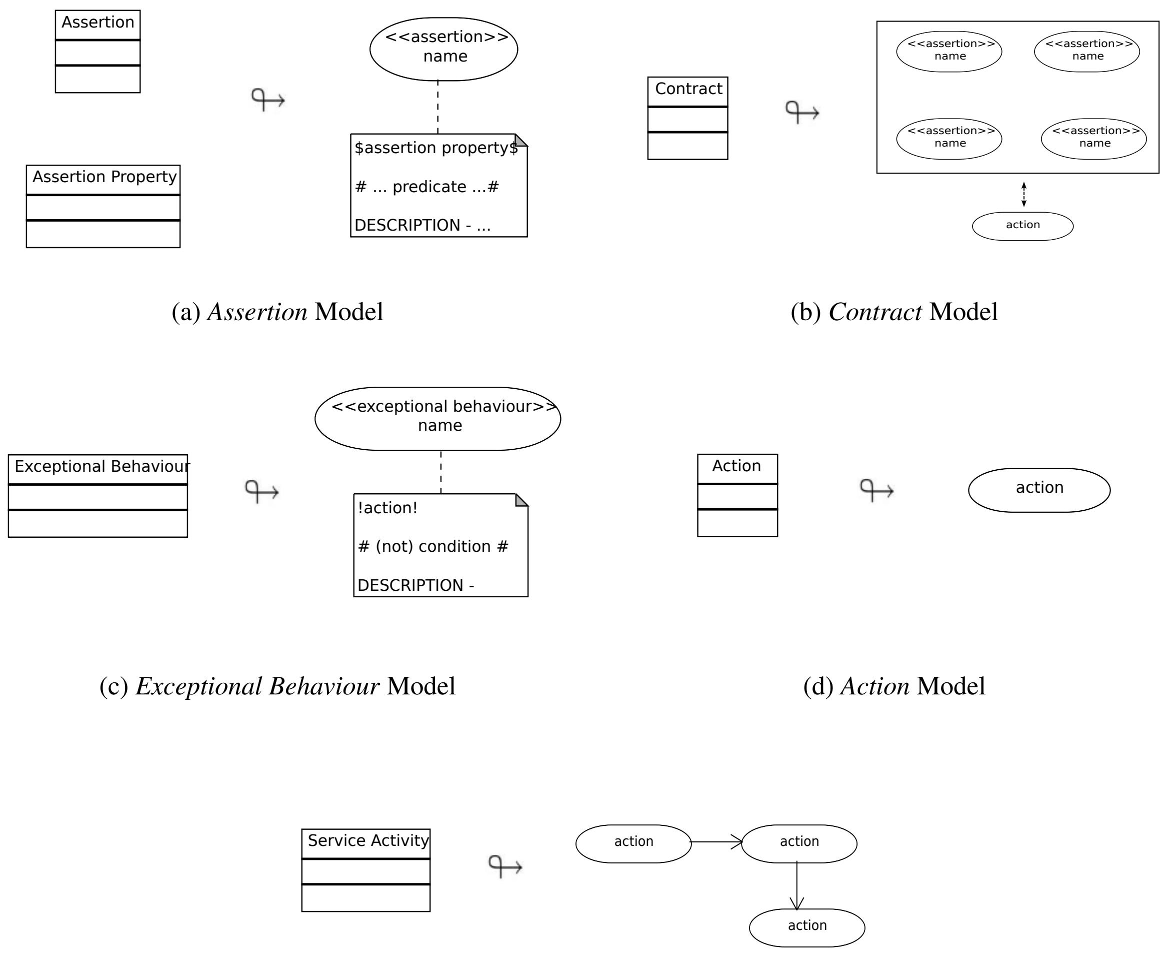


Figure 23: $\pi$-ServiceProcess Concepts (Meta-model).

the UML activity diagram, with an extension for CONTRACT design, which encapsulates several CONSTRAINTS modelled in $\pi$-*UseCase* model. In $\pi$-*ServiceProcess*, three main features are represented: (i) *service process*, (ii) *service activity* and (iii) *activity contract*. The service process is represented by the model itself. A service activity represents a behaviour that is part of the execution flow, and is identified in this model as an ACTION. A activity contract represents the NON-FUNCTIONAL REQUIREMENT behaviour that is also part of the execution flow of a service, and is identified in this model as an stereotyped activity («assertion»). All ASSERTION related with an ACTION compose a CONTRACT. From the CONTRACT and ASSERTION concepts is possible to define the interface specifications for each activity service, such as pre-conditions and post-conditions.

### 3.2.2.3 UML Concepts Representation

To formalize the concept of CONTRACT by rules, we use the definition of ASSERTION (figure 24a). An ASSERTION is represented by a stereotyped action («assertion»), and ASSERTION PROPERTIES through comments. An ASSERTION may be a *pre-condition*, *post-condition* or a *time restriction* over the service execution (ASSERTION PROPERTY). These information are represented as the comment *tags $ ... $*. The assertion predicate is modelled as *# ... #* tag. Figure 24a shows how these concepts are represented in UML.



(a) *Assertion* Model

(b) *Contract* Model

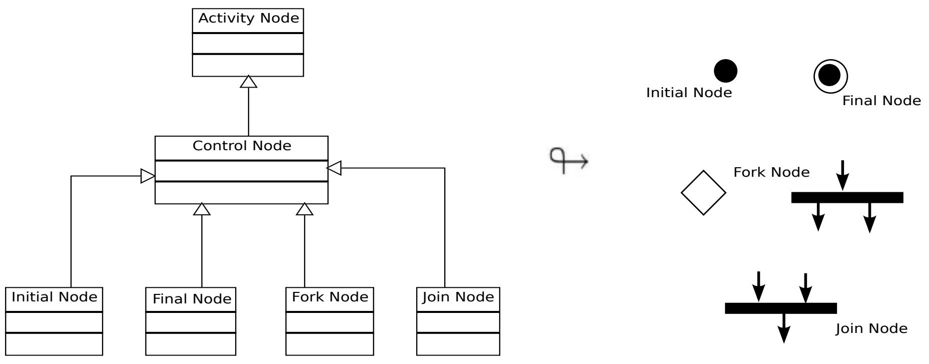
(c) *Exceptional Behaviour* Model

(d) *Action* Model

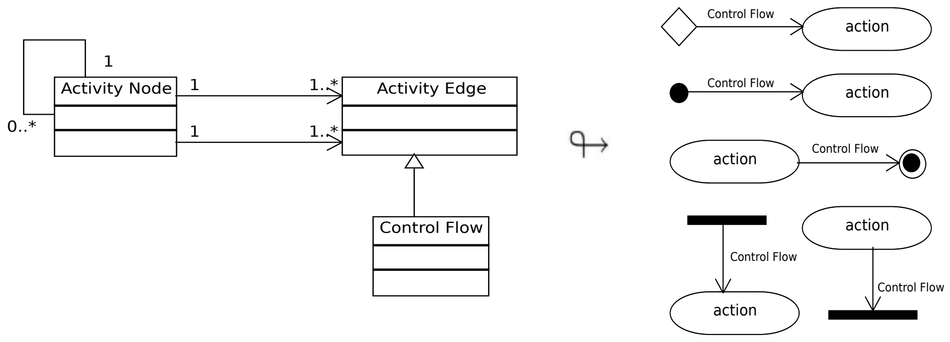
(e) *Service Activity* Model

Figure 24: $\pi$-*ServiceProcess* Model Representation

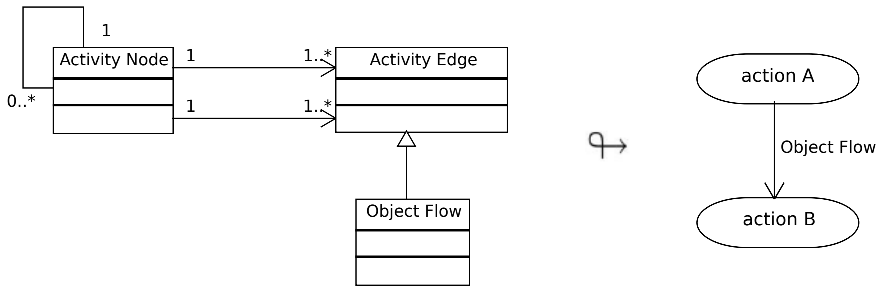
A set of ASSERTION composes a service CONTRACT related with an ACTION (figure 24b). An Assertion describes a value (*value constraint* from $\pi$-UseCase model) restriction, setting maximum and minimum limits, or business restrictions, setting conditions for function calls ((*business constraint* from $\pi$-UseCase model)). The business restrictions are defined over service functions. Every service contract may have an exception behaviour, if a constraint is not obeyed. For example, if the user card information are incorrect, the service is called again until the limit of 3 calls, otherwise, execution continues normally. In this case, the exception is the new call to the same service. For this, we also use the concept of EXCEPTIONAL BEHAVIOUR (figure 24c). A EXCEPTIONAL BEHAVIOUR is also a stereotyped action (`«exceptional behaviour»`) and its properties are modelled through comment tags. The condition to be verified is modelled as # . . . # tag. Figure 24a shows how these concepts are represented in UML. If the condition is *false*, the predicate becomes true, because of the expression negation



(a) *Control Nodes* Model



(b) *Activity Edge - Control Flow* Model



(c) *Activity Edge - Control Object* Model

Figure 25: $\pi$-*ServiceProcess* Model Representation (2)

(*not*). Thus the action is performed. The action to be performed is modelled as *! . . . !* tag. The ACTION concept is modelled in UML as the action component (figure 24d), and a SERVICE ACTIVITY represents a set of ACTION (figure 24e). We highlight the models in figures 24a, 24b and 24c, because they present how an assertion action, a contract, and a exceptional behaviour can be designed.

Figure25 shows the UML representation of the CONTROL NODES and ACTIVITY EDGES concepts in $\pi$-ServiceProcess meta-model. This representation follows the original UML specification. Figure 25a presents the CONTROL NODES and how its can be applied in a real modelling, and finally figures 25b and 25c show the ACTIVITY EDGES and its application.

#### 3.2.2.4 *To Publish Music* Process

Considering the example scenario, the contract based process of activities execution is shown in figure 26. The activities flow in figure 8 presents any restriction. Thus, it is necessary to represent each CONSTRAINT modelled in a more representative CONTRACT based service activity diagram.

Based on the concepts described and its representation in UML, we describe how to model a service process using the assertions as restrictions for each action. The difference of the model shown in figure 8 and the figure 26 is that restrictions are added on the actions in order to describe quality requirements.

Each USE CASE modelled in the use case diagram are refined into ACTIONS in the service process diagram. The set of CONSTRAINTS are transformed into ASSERTIONS and CONTRACTS. The *buy music* and *publish music* services (update Twitter and Facebook) has pre- and post-conditions assertions that are composed into a contract for each service. The *buy music* pre-conditions are: (i) verify if the User data are correct; (ii) if the User is already logged in Spotify; (iii) if bank account information are correct and; (iv) if there is enough money to make the payment. As post-condition, it ensures the complete transaction and verify if a notification were sent to the user and Spotify, about the payment authorization. There are four assertions for the *buy music* action, and each assertion has been detailed with the assertion property and predicate that must be verified. To *update* services, depending of each service, there may be different restrictions. As an example, a new verification of user data and message format is appropriate (maximum 140 characters), in case of Twitter. In the case of Facebook, it is required that the user is already logged on Spotify and these data are the same as Facebook. As post-condition, it ensures that the Facebook service send a notification of success. To update Twitter a pre-condition is required, while to update Facebook is necessary a pre-condition and a confirmation notice is modelled as post-condition. As a pre-condition for "*twitter update*" it is necessary that *(i)* the music format be correct and (*ii*) the twitter login and password be correct for the update.

After modeling of service process, its features, and the service contracts, it is necessary to model the service composition process and its policies. $\pi$SOD-M uses a service composition diagram to describe the third party services and each policy allied to each composition. In the

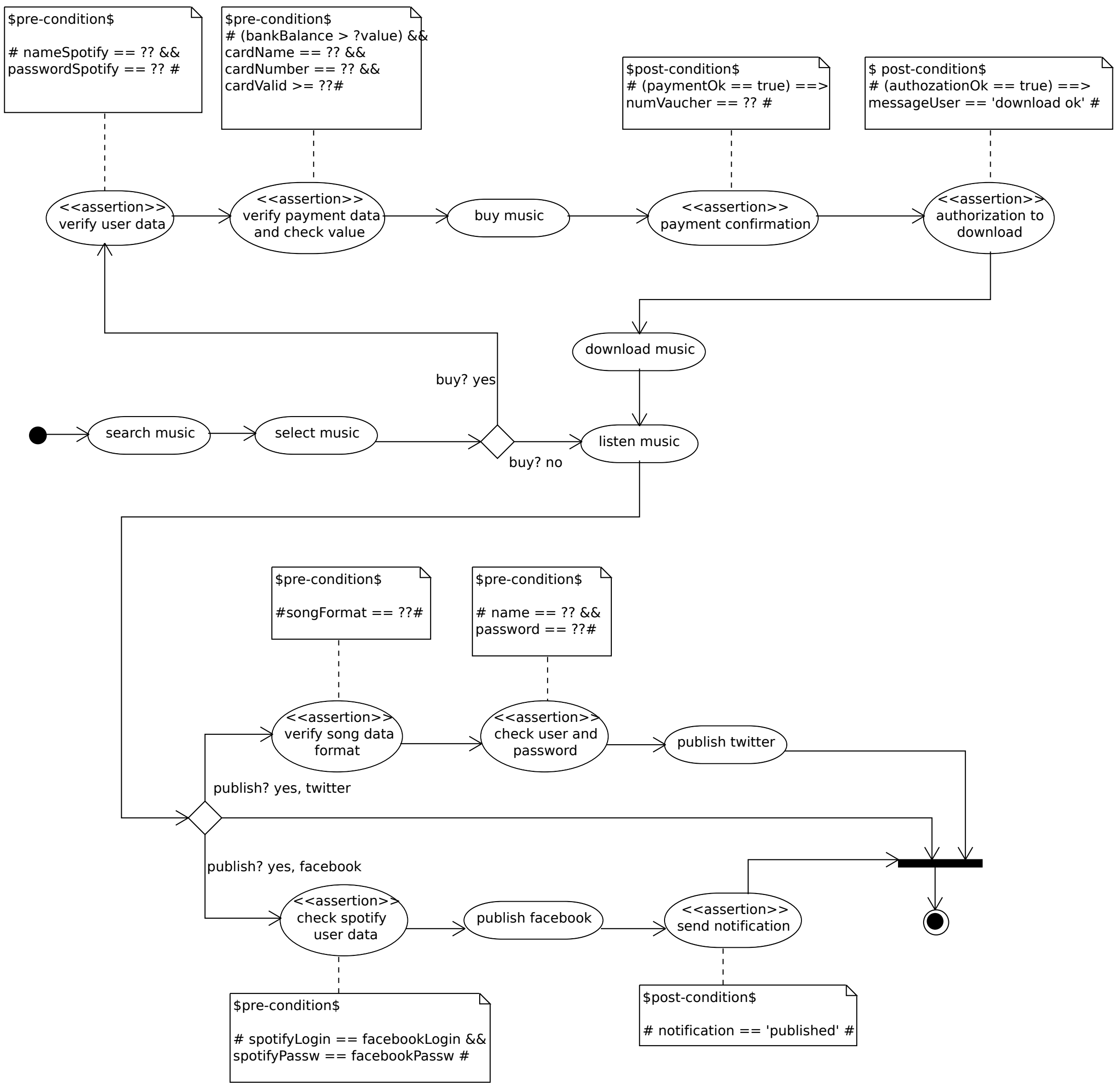

$pre-condition$
# nameSpotify == ?? && passwordSpotify == ?? #
$pre-condition$
# (bankBalance > ?value) && cardName == ?? && cardNumber == ?? && cardValid >= ??#
$post-condition$
# (paymentOk == true) ==> numVaucher == ?? #
$ post-condition$
# (authozationOk == true) ==> messageUser == 'download ok' #
<<assertion>> verify user data
<<assertion>> verify payment data and check value
buy music
<<assertion>> payment confirmation
<<assertion>> authorization to download
download music
buy? yes
search music
select music
listen music
buy? no
$pre-condition$
#songFormat == ??#
$pre-condition$
# name == ?? && password == ??#
<<assertion>> verify song data format
<<assertion>> check user and password
publish twitter
publish? yes, twitter
publish? yes, facebook
<<assertion>> check spotify user data
publish facebook
<<assertion>> send notification
$pre-condition$
# spotifyLogin == facebookLogin && spotifyPassw == facebookPassw #
$post-condition$
# notification == 'published' #

Figure 26: Service Process Model with Contract Definition - Complete Example.

next section we present this model and their respective policy concept.

### 3.2.3 $\pi$-ServiceComposition Model

The service composition model represents the workflow needed to execute a service or a set of them, but in a more detailed way, *i.e.*, identifying those entities that collaborate with the service process workflow, performing services that are necessary to execute the actions (in the business collaborators). Considering the workflow described in the $\pi$-*ServiceProcess* diagram, this model identifies the services and functions that correspond to each action in the business process.

Each action describes a fundamental behaviour unit that is part of a service activity and represents some behaviour in the system being modelled. The $\pi$-*ServiceComposition* model refines the concepts described in the $\pi$-*ServiceProcess* model, grouping contracts into policy (figure 23).

#### 3.2.3.1 $\pi$-*ServiceComposition* Diagram, Terms and Concepts

A $\pi$-*ServiceComposition* model represents service compositions and their related business collaborators (external services), policies, rules associated with a policy and the whole application process and functionalities.

This model is essential for the organization and modeling of services and their location. We refine the representation of packages described in the $\pi$-*UseCase* model into business collaborator. The set of use cases described in a package ($\pi$-*UseCase* model) will be represented by an external service functions.

Figure 27 shows how to relate the actions described in the $\pi$-*ServiceProcess* model with services of a specific business collaborator. A service is represented by an stereotyped action (*«service»*), and their connection is made by an edge (*Object Flow*). Thus each action described in the application workflow may be replaced by a service or a composition of them. If there is no relationship between the actions described in the application workflow (*External false* business collaborator) with external services, means that function is native to the application being developed.

The main properties described in the $\pi$-*ServiceComposition* model for the "*to publish music*" example are: (i) description of workflow application (*External false*), where the business collaborator *Application* points out that it is not an external workflow, (ii) the external services (*business collaborators*) and (iii) the application policies. Figures 27 and 28 presents how to model the $\pi$-*ServiceComposition* concepts (figure 29).

Figure 27 shows how each action is connected to an external service (through *Activity Edge - Object Flow*), and how to define the description of each business process, while Figure 28 shows each policy property and how it should be presented in the model.

Each business collaborator encompasses a set of services that can be performed by the application. These external services are used by the application must be described in this model, and not in the $\pi$-*ServiceProcess* model. The contracts described for each action are refined into policies, considering the non-functional requirement of each contract.

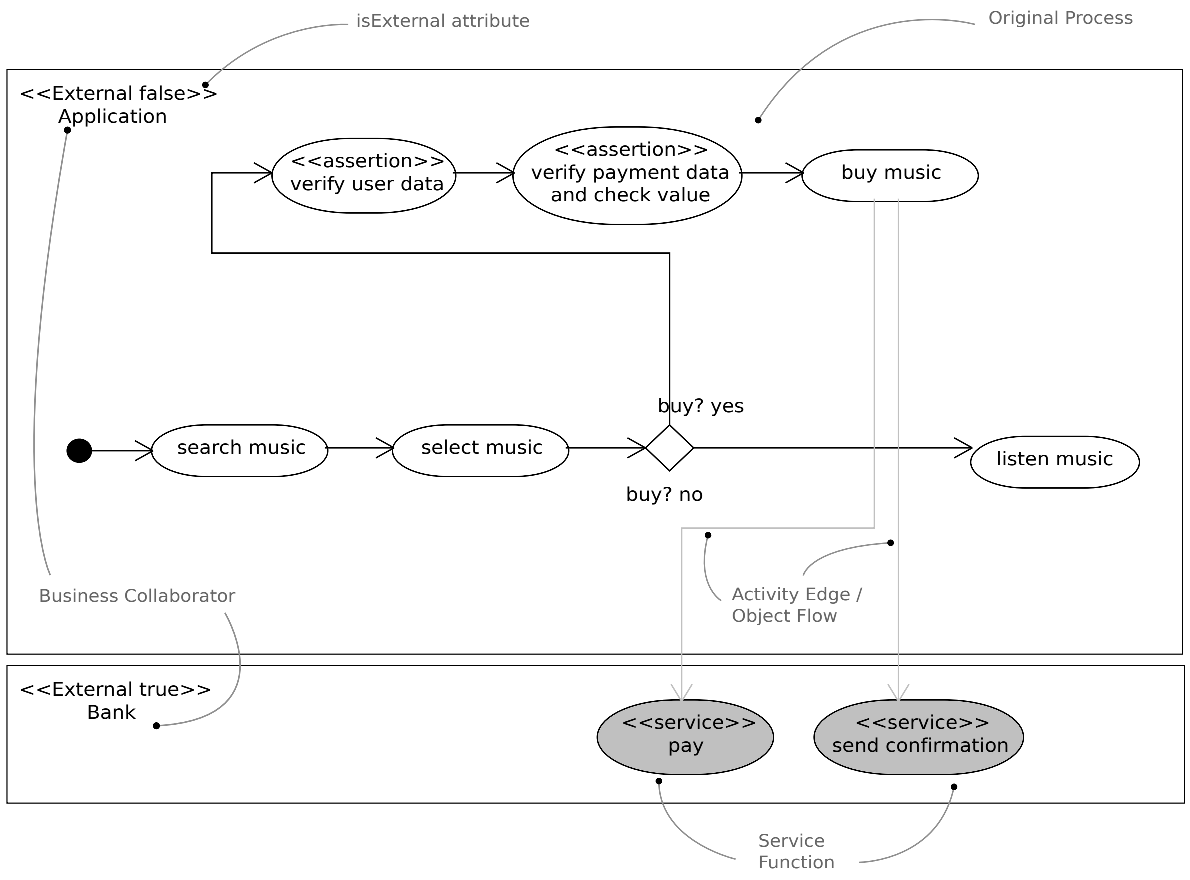


Figure 27: Business Collaborator Representation.

### 3.2.3.2 Meta-model

Considering the $\pi$SOD-M concepts (figure 6), the $\pi$-*ServiceComposition* meta-model (figure 29) describes the service compositions to be modelled and the policies related with each service composition. The $\pi$SOD-M concepts modelled in the $\pi$-*ServiceComposition* meta-model are: POLICY, RULE, VARIABLE, EXCEPTIONAL BEHAVIOUR, SERVICE ACTIVITY, ACTION, BUSINESS COLLABORATOR and NON-FUNCTIONAL REQUIREMENT.

$\pi$SOD-M proposes representing this model using the a extension of UML activity diagram technique. This extension describes the relationship of each ACTION for the respective SERVICE ACTIVITIES. Thus, as shown in figure 29, the meta-model includes typical modeling elements of the activity diagram such as ACTIVITY NODES, INITIAL NODES and FINAL NODES, DECISION NODES, etc., along with new elements defined by $\pi$SOD-M such as BUSINESS COLLABORATOR, SERVICE ACTIVITY and ACTION (see figures 29 and 31).

All $\pi$-*ServiceComposition* models describe the set of services and its relation with the service process and its activities that need to be performed. So, the external services are represented

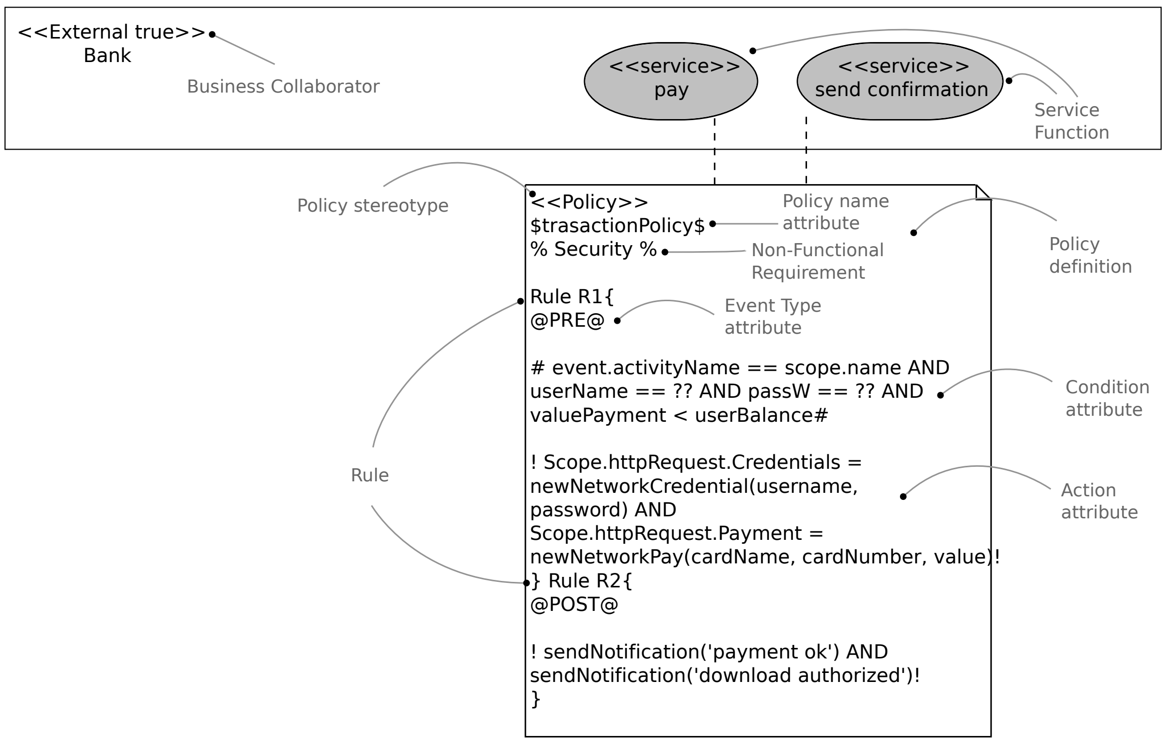

<<External true>>
Bank
Business Collaborator
<<service>>
pay
<<service>>
send confirmation
Service Function
Policy stereotype
Policy name attribute
Non-Functional Requirement
Policy definition
Event Type attribute
Condition attribute
Action attribute
Rule
<<Policy>>
$trasactionPolicy$
% Security %
Rule R1{
@PRE@
# event.activityName == scope.name AND userName == ?? AND passW == ?? AND valuePayment < userBalance#
! Scope.httpRequest.Credentials = newNetworkCredential(username, password) AND Scope.httpRequest.Payment = newNetworkPay(cardName, cardNumber, value)!
} Rule R2{
@POST@
! sendNotification('payment ok') AND sendNotification('download authorized')!
}

Figure 28: Policy Representation.

by the BUSINESS COLLABORATOR concept. All service composition may specify a POLICY as restriction. A POLICY is the set of CONTRACTS related with the service process expressed in the previous model ($\pi$-*ServiceProcess*).

Policy over service activities and business collaborators are not described in the original SOD-M method. The rule concept grouped into policy and exceptional behaviour make possible to model restrictions on service composition. Entities highlighted in the $\pi$-*ServiceComposition* (figure 29) meta-model are the difference from the original SOD-M service composition model.

A BUSINESS COLLABORATOR element represents those entities that collaborate in the business processes by performing some of the required actions. They are graphically presented as a partition in the activity diagram. A collaborator can be either internal or external to the system being modelled. When the collaborator of the business is external to the system, the attribute *IsExternal* of the collaborator is set to *true*.

ACTION, a kind of EXECUTABLE NODE, are represented in the model as an activity. Each action identified in the model describes a fundamental behaviour unit which represents some type of transformation or processing in the system being modelled. There are two types of actions: i) a Web Service (attribute Type is *WS*); and ii) a simple operation that is not supported by a Web Service, called an ACTIVITY OPERATION (attribute Type is *AOP*).

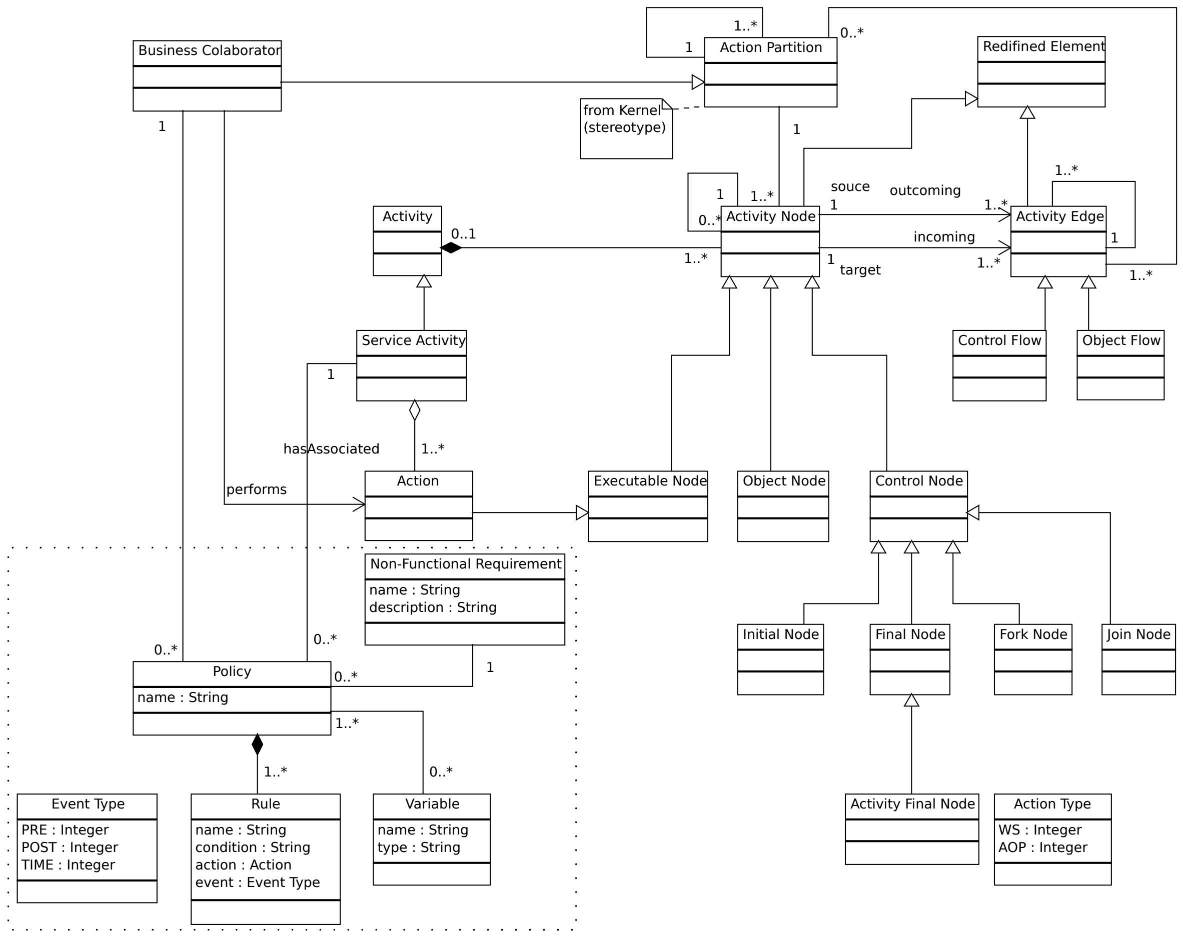


Figure 29: $\pi$-*ServiceComposition* Concepts (Meta-model.

The SERVICE ACTIVITY element is a composed activity that must be carried out as part of a business service and is composed of one or more executable nodes.

The policy based service composition model refines the concept contract of service process model. A policy assemble a set of contracts and can be applied to more than one activity. The restriction on service composition model is marked with the stereotype «`policy`».

An *Policy* groups a set of rules. It describes global variables and operations that can be shared by the rules and that can be used for expressing their Event and Condition parts. An *Policy* is associated to the elements BUSINESS COLLABORATOR, SERVICE ACTIVITY and, ACTION of the $\pi$-ServiceComposition meta-model (see figure 29) .

Instead of programming different protocols within the application logic, we propose to include the modeling of non-functional requirements like transactional behaviour, security and adaptability at the early stages of the services' composition engineering process.

#### 3.2.3.3 UML Concepts Representation

The representation described in figure 30 presents how the user of the methodology's user apply the concepts described in the $\pi$-*ServiceComposition* meta-model. In the specific case of $\pi$-*ServiceComposition* meta-model, those concepts will be used to describe how BUSINESS COLLABORATOR, ACTION, SERVICE ACTIVITY, POLICY AND RULES can be specified. Figure 30 also describes how the proposed concepts can be modelled in real system development cases. All $\pi$-*ServiceComposition* based models (presented in figure 30) must follow the meta-model description and its constraints.

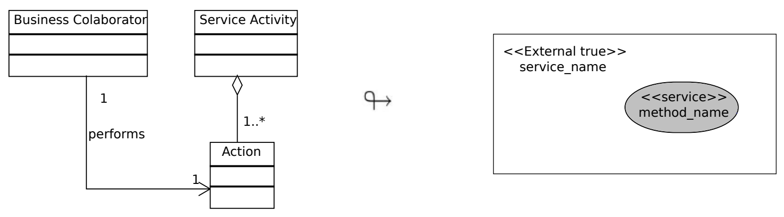
Figure 30a presents the representation for the relation between ACTION and BUSINESS COLLABORATOR. As a SERVICE ACTIVITY are modelled as a set of ACTIONS (service functions), they are represented as *«External true»* BUSINESS COLLABORATOR. It means that an external service is invoked by the application's function. Each BUSINESS COLLABORATOR can represents an external service. Figure 30b presents the representation for the relation between ACTION, POLICY and RULE. Figure 30c presents the representation of the original *Service Process* associated to the *«External false»* BUSINESS COLLABORATOR and figure 30d presents how to associate the *Service Process*' ACTION with real services.

Notice that all $\pi$*ServiceComposition* meta-model concepts can be represented in the real application model, as we illustrate next using the scenario example.

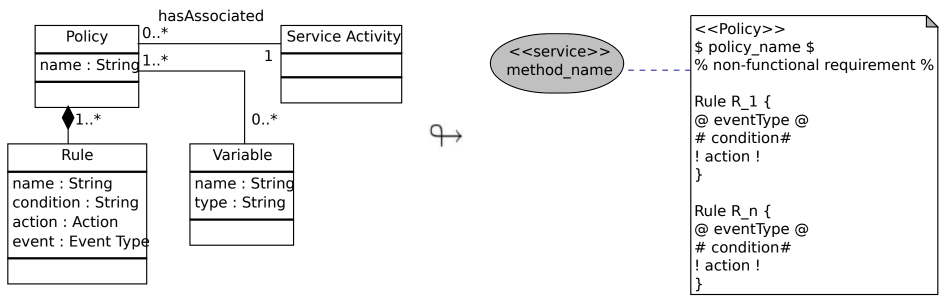
#### 3.2.3.4 *Publish Music* Service Composition

To illustrate the use of the $\pi$-*ServiceComposition* model we used it for defining the policy based composition model of the example scenario (see figure 31). There are four external BUSINESS COLLABORATORS, they are: *Bank, Spotify, Twitter* and *Facebook.* The model also shows the business process of the application that consists of six SERVICE ACTIVITIES (see figure 8): *search music, select music, buy music, download music, listen music* and *publish music*. Note that the action *publish music* of the application calls the actions of two service collaborators namely Facebook and Twitter, and the action *buy music* of the application calls two actions of the service collaborator Bank.

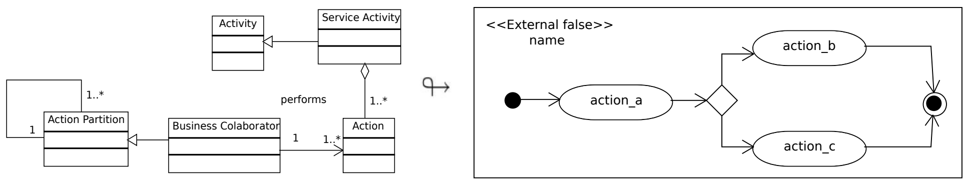
The *Facebook* and *Twitter* services require authentication protocols in order to execute methods that will read and update the users' space. A call to such services must be part of the authentication protocol required by these services. In this example we associate two authentication policies, one for the open authentication protocol, represented by the class Twitter AuthPolicy that will be associated to the activity UpdateTwitter (see figure 31). In the same way,



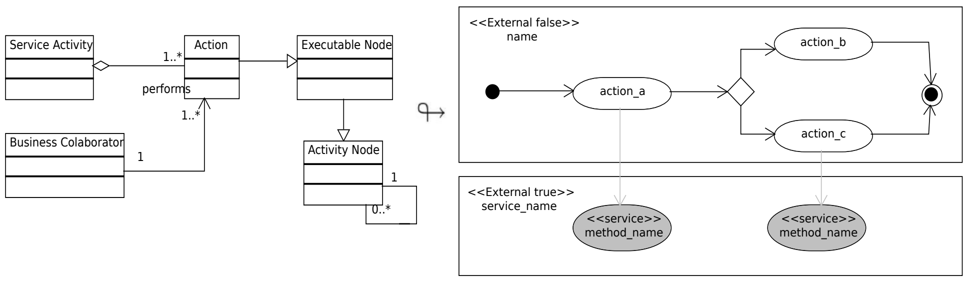
(a) *Service Activity* and *Business Collaborator* Model



(b) *Policy* and *Rule* Model



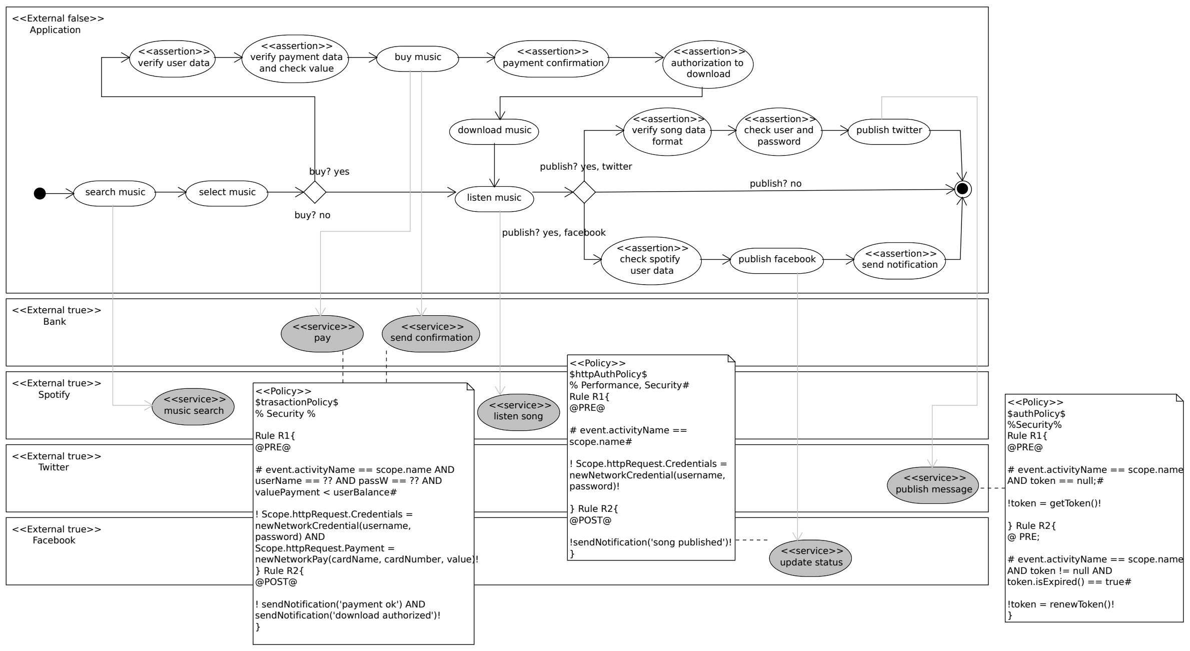
(c) *Business Collaborator - External false* Model



(d) *Business Collaborator - External true* Model

Figure 30: $\pi$-*ServiceComposition* Representation Models

the class Facebook HTTPAuthPolicy, for the http authentication protocol will be associated to the activity UpdateFacebook. OAuth implements the open authentication protocol. As shown in figure 31, the *Policy* has a variable Token that will be used to store the authentication token provided by the service. This variable is imported through the library Auth.Token. The *Policy* defines two rules, both can be triggered by events of type ActivityPrepared: (i) if no token has been associated to the variable token, stated in by the condition of rule $R_1$, then a token is ob-



<<External false>> Application
<<assertion>> verify user data
<<assertion>> verify payment data and check value
buy music
<<assertion>> payment confirmation
<<assertion>> authorization to download
download music
<<assertion>> verify song data format
<<assertion>> check user and password
publish twitter
search music
select music
buy? yes
buy? no
listen music
publish? yes, twitter
publish? no
publish? yes, facebook
<<assertion>> check spotify user data
publish facebook
<<assertion>> send notification
<<External true>> Bank
<<service>> pay
<<service>> send confirmation
<<External true>> Spotify
<<service>> music search
<<service>> listen song
<<External true>> Twitter
<<service>> publish message
<<External true>> Facebook
<<service>> update status

```
<<Policy>>
$trasactionPolicy$
% Security %

Rule R1{
@PRE@

# event.activityName == scope.name AND
userName == ?? AND passW == ?? AND
valuePayment < userBalance#

! Scope.httpRequest.Credentials =
newNetworkCredential(username,
password) AND
Scope.httpRequest.Payment =
newNetworkPay(cardName, cardNumber, value)!
} Rule R2{
@POST@

! sendNotification('payment ok') AND
sendNotification('download authorized')!
}
```

```
<<Policy>>
$httpAuthPolicy$
% Performance, Security#
Rule R1{
@PRE@

# event.activityName ==
scope.name#

! Scope.httpRequest.Credentials =
newNetworkCredential(username,
password)!

} Rule R2{
@POST@

!sendNotification('song published')!
}
```

```
<<Policy>>
$authPolicy$
%Security%
Rule R1{
@PRE@

# event.activityName == scope.name
AND token == null;#

!token = getToken()!

} Rule R2{
@ PRE;

# event.activityName == scope.name
AND token != null AND
token.isExpired() == true#

!token = renewToken()!
}
```

Figure 31: Service Composition Model with Policy Definition - Spotify Music Service.

tained (action part of $R_1$); (ii) if the token has expired, stated in the condition of rule $R_2$, then it is renewed (action part of $R_2$). Note that the code in the actions profits from the imported Auth.Token for transparently obtaining or renewing a token from a third party.

HTTP-Auth implements the HTTP-Auth protocol. As shown in figure 31, the *Policy* imports an http protocol library and it has two variables username and password. The event of type ActivityPrepared is the triggering event of the rule $R_1$. On the notification of an event of that type, a credential is obtained using the username and password values. The object storing the credential is associated to the scope, i.e., the activity that will then use it for executing the method call.

Thanks to rules and policies it is possible to model and associate non-functional properties to services' compositions and then generate the code. For example, the atomic integration of information retrieved from different social network services, automatic generation of an integrated view of the operations executed in different social networks or for providing security in the communication channel when the payment service is called.

Back to the definition process of a SIS, once the *Policy* based services' composition model has been defined, then it can be transformed into a model (i.e., $\pi$-PEWS model, Section 3.3) that can support then executable code generation.

## 3.3 $\pi$-PEWS Platform Specific Models

Platform specific models (PSMs) are used to combine the specifications in the PIM with the details of the chosen implementation platform. PSM models are used to implement the system (generating code automatically), and they must provide all information needed to build the system and its components. A PSM model can also run as a model to be used for further refinements by others PSM models.

$\pi$SOD-M platform model proposed at PSM level combines specific details of the service-based platforms. We use a $\pi$-PEWS language meta-model for representing the specification of a service composition on the language. A model can be then used to generate the corresponding code. $\pi$-PEWS [80] is a extension of the PEWS language that provides constructs for specifying policies for services through contracts clauses (see Appendix C).

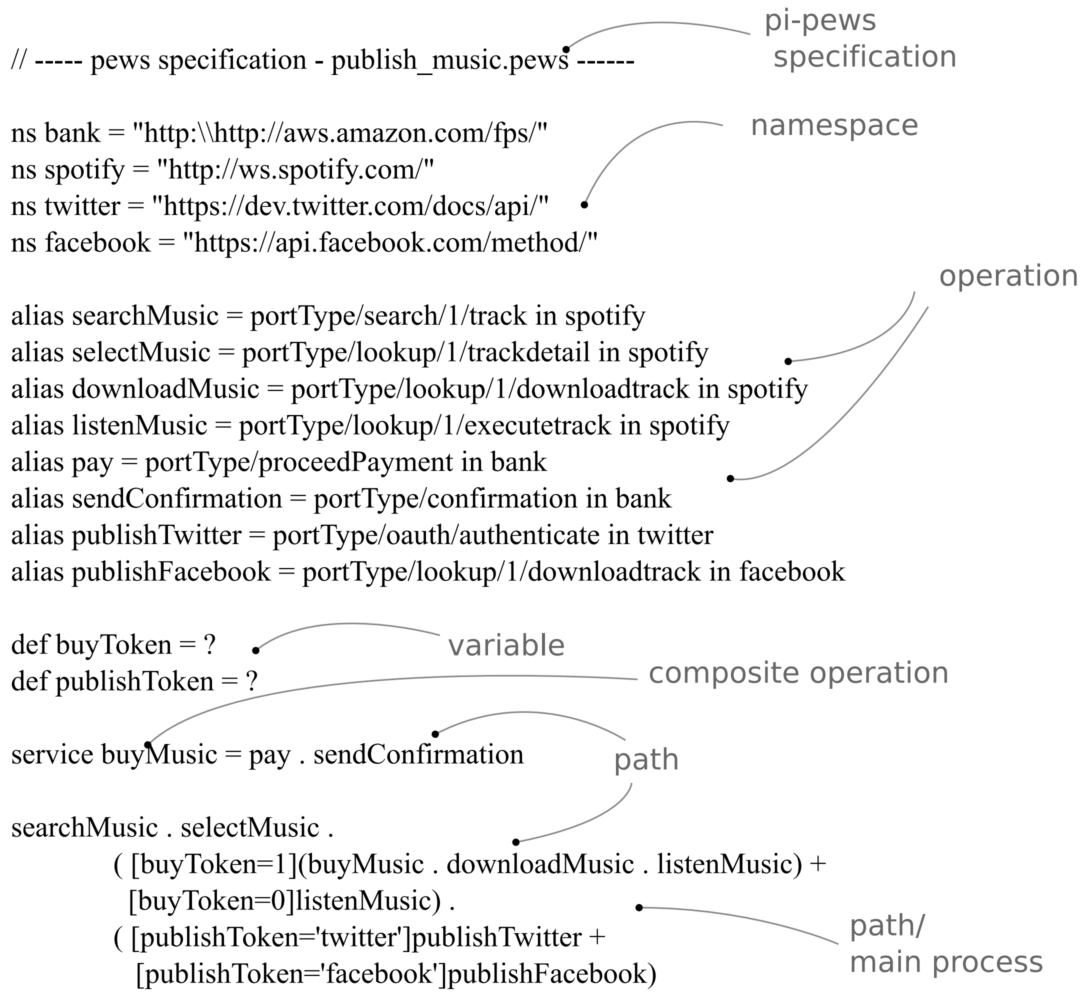
### 3.3.1 $\pi$-PEWS Specification, Terms and Concepts

A *$\pi$-PEWS* model represents the system specification. This model is essential for the system specification, detailing the services, compositions and system's constraints. A *$\pi$-PEWS* specification describes the behavioural aspects of the system.

We extended the PEWS language [80] to support the notion *policy* through contracts definitions (see figure 33 and appendix C). The extension does not modify the syntax of existing PEWS programs, but complements it by specifying (non-functional) restrictions in a separate block. This feature is intended to enhance reusability and allows the programmer to adequate a program to specific requirements of the application being developed. This means that the program developer can reuse the control part of the program and add application-specific requirements as contract or time constraints.

Behavioural web service interface languages describe not only the input/output interface of a web service but also the expected behaviour of its components. This behaviour can be specified as a workflow, defining the *order* in which the components of a service will be executed, so that the workflow specifies the functional behaviour of the compound web service (figures 32 and 33).

Additionally, we can specify non-functional behaviour for a service; *i.e*, to impose some additional restrictions which are separated from the application's workflow. This can be done by using the notion of *contract* to be added to each particular instantiation of the PEWS program. The result of this is to allow a given service (workflow) to have different restrictions in different



```
// ----- pews specification - publish_music.pews ------

ns bank = "http:\\http://aws.amazon.com/fps/"
ns spotify = "http://ws.spotify.com/"
ns twitter = "https://dev.twitter.com/docs/api/"
ns facebook = "https://api.facebook.com/method/"

alias searchMusic = portType/search/1/track in spotify
alias selectMusic = portType/lookup/1/trackdetail in spotify
alias downloadMusic = portType/lookup/1/downloadtrack in spotify
alias listenMusic = portType/lookup/1/executetrack in spotify
alias pay = portType/proceedPayment in bank
alias sendConfirmation = portType/confirmation in bank
alias publishTwitter = portType/oauth/authenticate in twitter
alias publishFacebook = portType/lookup/1/downloadtrack in facebook

def buyToken = ?
def publishToken = ?

service buyMusic = pay . sendConfirmation

searchMusic . selectMusic .
          ( [buyToken=1](buyMusic . downloadMusic . listenMusic) +
            [buyToken=0]listenMusic) .
          ( [publishToken='twitter']publishTwitter +
            [publishToken='facebook']publishFacebook)
```

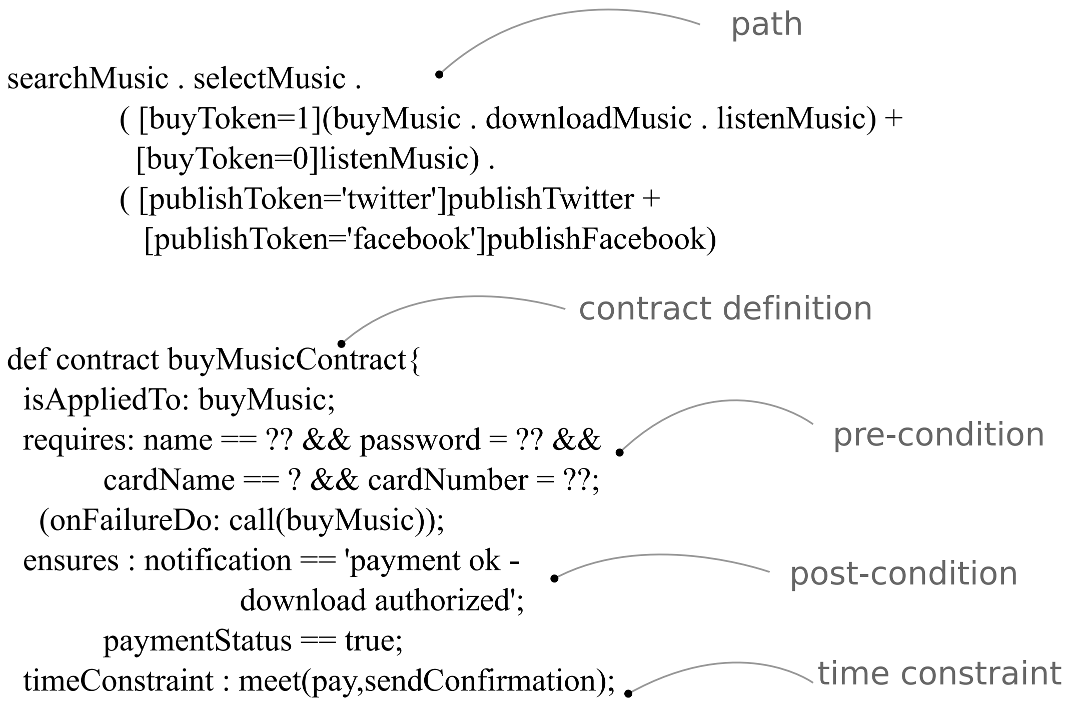
Figure 32: $\pi$-PEWS Specification Representation.

contexts.

The policy model proposed here follows the main ideas presented in [42, 81], where contracts are added to a given composition, as a separate concern. The properties defined by a contract should be verified at runtime. Recovery actions, defined by the contract, are to be performed in case of failure of the contract's conditions. Recovery actions are defined by the contract itself.

## 3.3.2 Meta-model

The idea of the $\pi$-PEWS meta-model is based on the services' composition approach provided by the language PEWS [28, 80] (*Path Expressions for Web Services*). Figure 34 presents the $\pi$-PEWS meta-model consisting of classes representing:



```
searchMusic . selectMusic .
        ( [buyToken=1](buyMusic . downloadMusic . listenMusic) +
          [buyToken=0]listenMusic) .
        ( [publishToken='twitter']publishTwitter +
          [publishToken='facebook']publishFacebook)

def contract buyMusicContract{
  isAppliedTo: buyMusic;
  requires: name == ?? && password = ?? &&
        cardName == ? && cardNumber = ??;
   (onFailureDo: call(buyMusic));
  ensures : notification == 'payment ok -
                       download authorized';
        paymentStatus == true;
  timeConstraint : meet(pay,sendConfirmation);
```

Figure 33: $\pi$-PEWS Contract Representation.

- A services' composition: NAMESPACE representing the interface exported by a service, OPERATION that represents a call to a service method, COMPOSITE OPERATION, and OPERATOR for representing a services' composition and PATH representing a services' composition. A PATH can be an OPERATION or a COMPOSITE OPERATION denoted by an identifier. A COMPOSITE OPERATION is defined using an OPERATOR that can be represent sequential ( . ) and parallel ( $\|$ ) composition of services, choice ( $+$ ) among services, the sequential ($*$) and parallel ($\{\ldots\}$) repetition of an operation or the conditional execution of an operation ($[C]S$).

- *Policies* that can be associated to a services' composition: A-POLICY, RULE, EVENT, CONDITION, ACTION, STATE, and SCOPE.

As shown in the diagram an POLICY is applied to a SCOPE that can be either an OPERATION (e.g., an authentication protocol associated to a method exported by a service), an OPERATOR (e.g., a temporal constraint associated to a sequence of operators, the authorized delay between reading a song title in Spotify and updating the walls must be less then 30 seconds), and a PATH (e.g., executing the walls' update under a strict atomicity protocol – all or noting). It groups a set of ECA rules, each rule having a classic semantics, i.e, *when an event of type E occurs if condition C is verified then execute the action A*. Thus, an *Policy* represents a set of reactions to be possibly executed if one or several triggering events of its rules are notified.

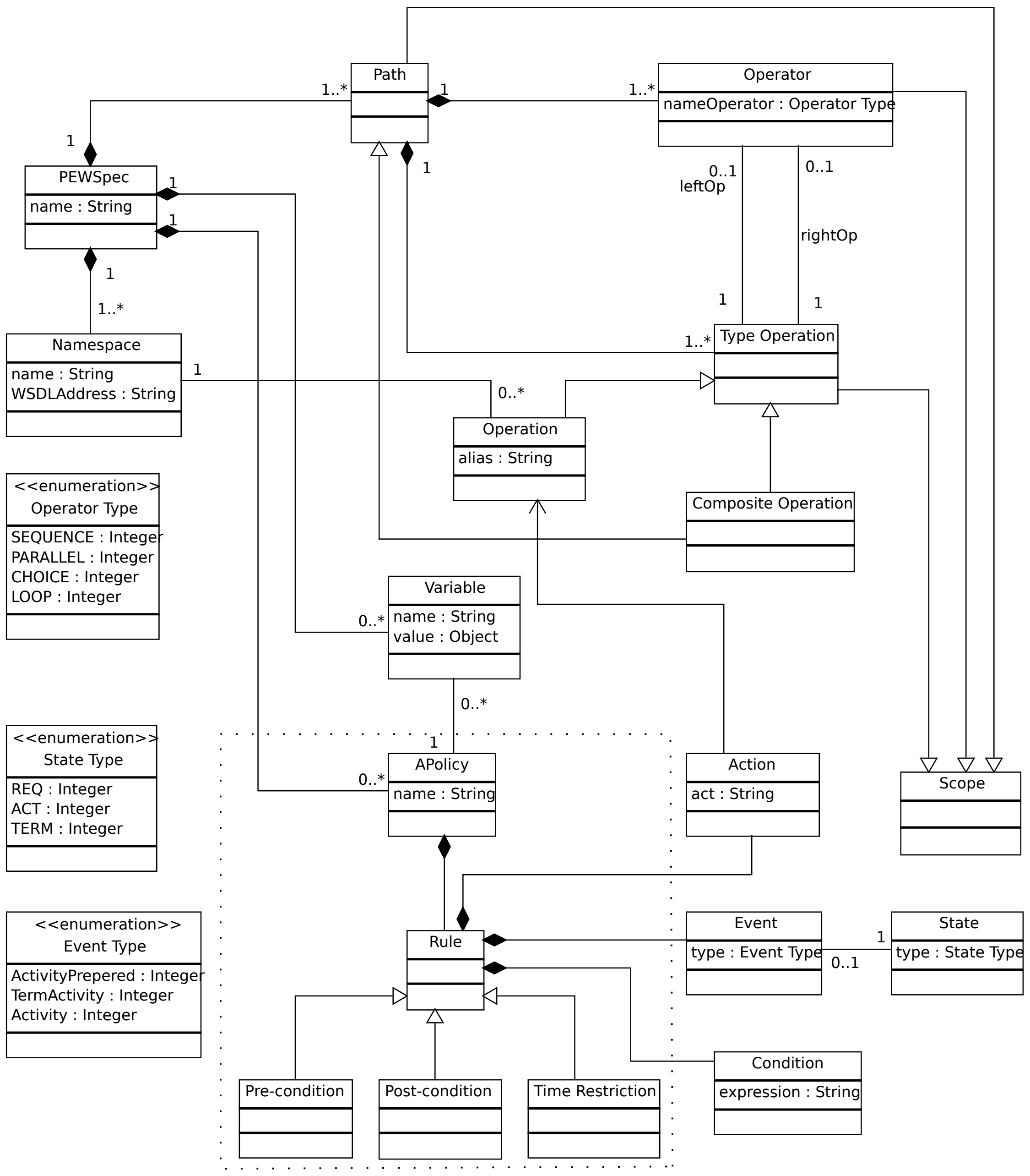


Figure 34: π-PEWS Meta-model.

- The class SCOPE represents any element of a services' composition (i.e., operation, operator, path).

- The class POLICY represents a recovery strategy implemented by ECA rules of the form EVENT - CONDITION - ACTION. A *Policy* has variables that represent the view of the execution state of its associated scope, that is required for executing the rules. The value of a variable is represented using the type VARIABLE. The class POLICY is specialized for defining specific constraints, for instance authentication *Policies*.

Given a $\pi$-ServiceComposition model of a specific service-based application (expressed according to the $\pi$-ServiceComposition meta-model), it is possible to generate its corresponding $\pi$-PEWS model by using transformation rules. The following Section describes the transformation rules between the $\pi$-ServiceComposition and $\pi$-PEWS meta-models of our method.

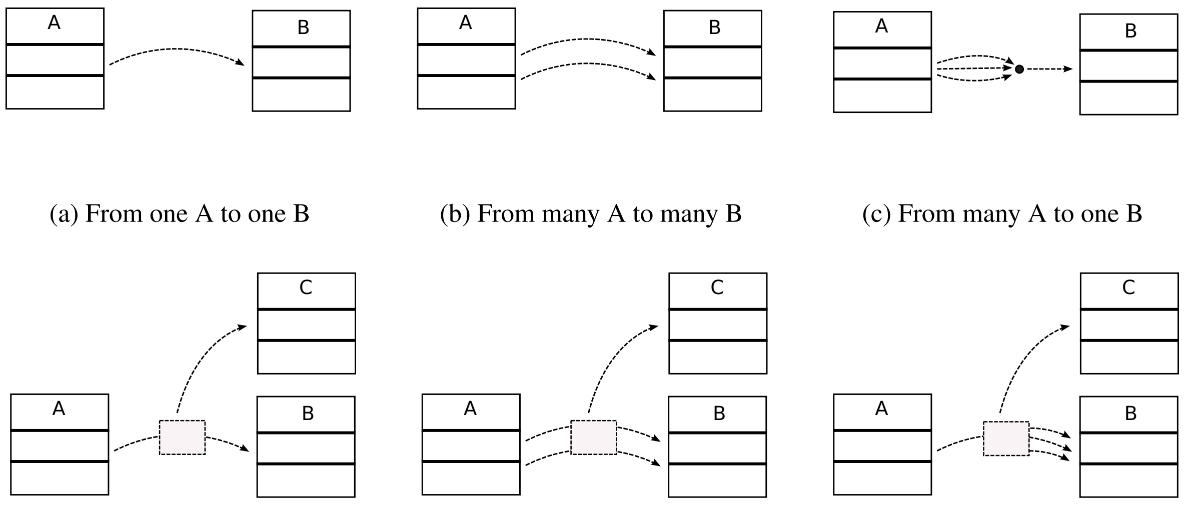
The $\pi$-PEWS language (extension of PEWS) is described in the Appendix C. The generated code by $\pi$SOD-M, as end product, is based on the meta-model shown in figure 29 and the language syntax.

## 3.4 Model Transformations

A model transformation usually specifies which models are acceptable as input, and if appropriate what models it may produce as output, by specifying the meta-model to which model must conform. Model transformations can be seen as processes that take models as input and produces models as output. There is a wide variety of kinds of model transformation and uses of them, which differ in their inputs and outputs and also in the way they are expressed [67]. A model transformation is also a way of ensuring that a family of models is consistent, in a precise sense which the software engineer can define. The aim of using a model transformation is to save effort and reduce errors by automating the building and modification of models where possible [49].

The $\pi$SOD-M model transformations occur in the two levels, PIM and PSM, we defined: PIM-to-PIM and PIM-to-PSM transformations. Transformations in the same level are considered "horizontal transformations". Transformations between different levels are called "vertical transformations". There are 3 model transformations defined in $\pi$SOD-M: $\pi$-UseCase to $\pi$-ServiceProcess; $\pi$-ServiceProcess to $\pi$-ServiceComposition; and $\pi$-ServiceComposition to $\pi$-PEWS.

All $\pi$SOD-M transformation rules are described in natural language. These transformations ensure consistency between the concepts being refined and processed at different levels. Figure 35 presents the set of rules we defined to apply for each type of transformation. The entities of the left-hand side represent the source model, and those on the right-hand side represent the target model. Figure 35a shows the transformation rule in which a single source entity is transformed in a target model entity. Figure 35b describes a many to many entity transformation: a set of objects from the source model is transformed into a set of objects in the target model. Figure 35c presents a many to one transformation rule: many objects in the source model it will be transformed into a single object element in the target model. The next 3 rules are advanced



(a) From one A to one B (b) From many A to many B (c) From many A to one B

(d) From one A to one B, and one C (e) From many A to many B, and one C (f) From many A to many B, and one C

Figure 35: Entities' Model Transformation Rules

rules, which transform single element in the source model into a set of different elements in the target model. In figure 35d, a source element is transformed in two different ones, *e.g.* from a "*A*" element, it generates a "*B*" and a "*C*" element in the target model. In figure 35e, a set of source elements is transformed into a set of elements "*B*" and one "*C*" element in the target model. Finally, figure 35f presents the rule which transforms source elements into a set of elements "*B*" and one element "*C*" in the target model.

### 3.4.1 From $\pi$-UseCase to $\pi$-ServiceProcess

The PIM to PIM transformation process between $\pi$-UseCase and $\pi$-ServiceProcess (Table 5) models defines how the application requirements are represented as a service process. At this level, services and general functions are represented as simple or composite use cases.

Every USE CASE is transformed into an ACTION in the $\pi$-*ServiceProcess* model, and every EXTEND association identified in the $\pi$-*UseCase* model is transformed into a FORK NODE. If the extend association has only one USE CASE, the fork will present the ACTION as an alternative flow, and later, both flows will meet. If the extend association has several source USE CASE, the fork will present different ACTIONS as mutual alternatives flows, and later, all these flows will meet. A ACTIVITY SERVICES (from $\pi$-ServiceProcess meta-model) consists of a composition of one or more ACTIONS. ACTION can be viewed as a method of the system application. Thus, services are represented as a set of functions in the $\pi$-*ServiceProcess* meta-model, because CONTRACTS modelled are associated to functions (ACTIONS).

All CONSTRAINTS in the source model ($\pi$-*UseCase model*) are transformed into a CONTRACT. A CONTRACT is represented as a set of CONSTRAINTS. Each constrain description is directly transformed into ASSERTION. A set of CONSTRAINT of a USE CASE or a COMPOSITE USE CASE is transformed into a set of ASSERTION used to define a CONTRACT. All USE CASES are associated to a NON-FUNCTIONAL REQUIREMENT and all CONSTRAINTS are associated with a NON-FUNCTIONAL ATTRIBUTE. Thus, CONSTRAINT representing the same USE CASE is transformed into a CONTRACT. A CONTRACT can have exceptional cases. The negation of the CONSTRAINT predicate are treated as EXCEPTIONAL BEHAVIOUR. In this particular case, a EXCEPTIONAL BEHAVIOUR will be generated in the target model.

The CONSTRAINT TYPE can have different transformations, depending of the following values:

- If the value of CONSTRAINT TYPE is *BUSINESS*, it will be transformed into none ASSERTION;
- If the value of CONSTRAINT TYPE is *VALUE* with the ASSERTION *isExceptionalBehaviour* attribute setted to *false*, it will be transformed into one ASSERTION;
- If the value of CONSTRAINT TYPE is *VALUE* with the ASSERTION *isExceptionalBehaviour* attribute setted to *true*, it will be transformed into a EXCEPTIONAL BEHAVIOUR;

Given the $\pi$-*UseCase* model, to all CONSTRAINT entity related with a USE CASE, there is a CONTRACT that compounds a set of ASSERTIONS entity (figure 36a) and the USE CASE is refined in a service ACTION.

*Example:* Considering the scenario example presented in Section 3.1.4, the transformation of the "*listen music*" USE CASE is transformed into a service ACTION. This action is a Spotify service function that can be invoked to play the music. For "*publish music*" USE CASE, all CONSTRAINS are transformed in a set of ASSERTIONS that are grouped in a CONTRACT ("*publishMusicContract*"). The "*publishMusicContract*" is related with the "*publishMusic*" ACTION. □

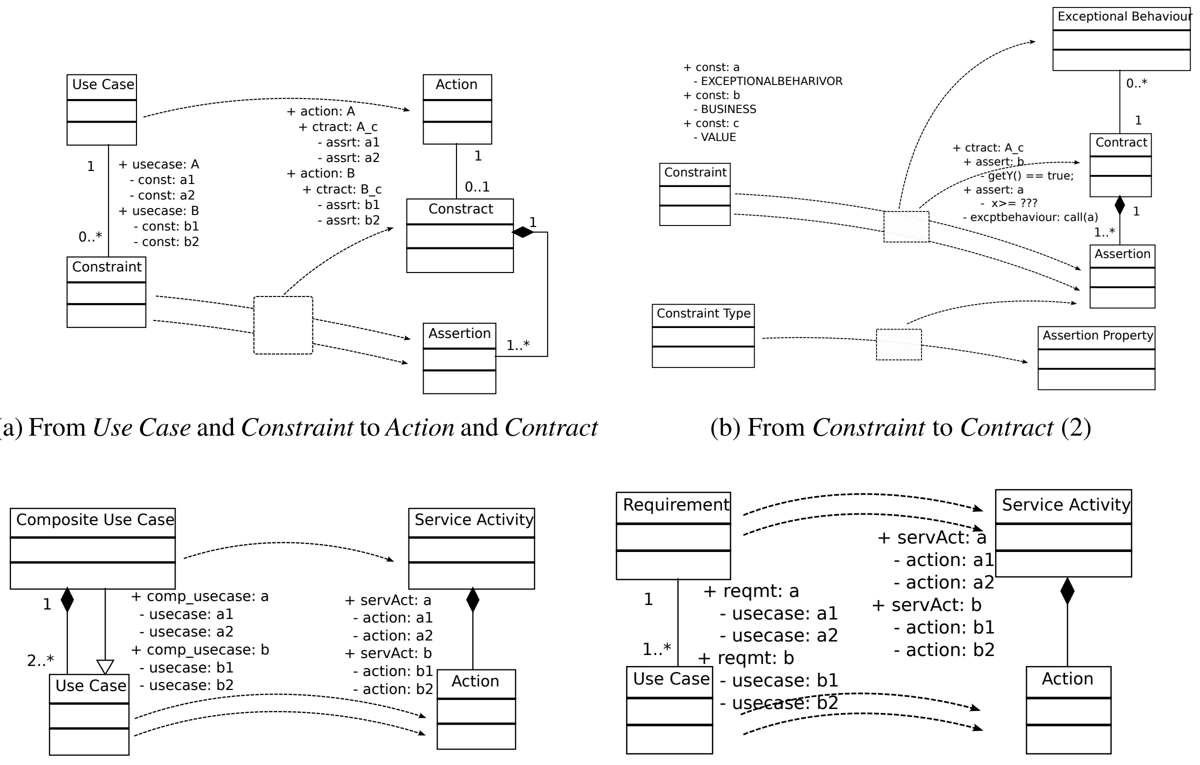
A CONSTRAIN transformation means that from CONSTRAINT and CONSTRAINT TYPE entities are generated detailed CONTRACT information, that are refined into EXCEPTIONAL BEHAVIOURS and ASSERTIONS entities (figure 36b). CONSTRAINT is related with a CONSTRAINT TYPE, and there are different cases for the transformation of this concept. A *Business*

Table 5: Transformation Rules: From $\pi$-UseCase to $\pi$-ServiceProcess

| Source<br>$\pi$-UseCase meta-model | Mapping Rules | Target<br>$\pi$-ServiceProcess meta-model |
|---|---|---|
| Constraint | - All **Constraints** in the source model will be transformed into a **Assertion** in the target model.<br>All **Use Cases** are associated to a **Non-Functional Requirement** and all **Constraints** are associated with a **Non-Functional Attribute**. Thus, **Constraint** representing the same Use Case, is transformed in a **Contract**.<br>- A **Contract** can also have a exceptional cases. The negation of the **Constraint** predicate are treated as the type *EXCEPTIONALBEHAVIOUR*. Thus, all the **Assertions** related to this **Constraint**, in this particular case, will be transformed in a **Exceptional Behaviour**.<br>- The set of ASSERTIONS of a USE CASE are associated with a CONTRACT<br>- The CONTRACT name is formed by "<useCaseName>" + "Contract" token. | Assertion, Contract,<br>Exceptional Behaviour |
| Constraint Type | - The **Constraint Type** in source model can have the following transformation:<br>● **BUSINESS** type is transformed into none **Assertion**;<br>● **VALUE** with the **Assertion** *isExceptionalBehaviour* attribute setted to false is transformed into a **Assertion** in the target model;<br>● **EXCEPTIONALBERAVIOR** with the **Assertion** *isExceptionalBehaviour* attribute setted to true is transformed into a **Exceptional Behaviour** in the target model.<br>- Both, the value and exceptional restriction are based on in accordance with the values of **Assertions**<br>- **Constraint Type** defines the type of restriction on a **Use Case**. | Assertion<br>Exceptional Behaviour |
| Non-Functional Attribute | Every **Non-Functional Attribute** associated to an element (**Constraint** and **Non-Functional Requirements**) in the source model becomes a **Non-Functional Attribute** associated to the corresponding element (**Contract**) in the target model. | Non-Functional Attribute |
| Use Case | For every **Use Case** there will be a **Action** in the service delivery process model describing the Business Service | Action |
| Extend (Use Case) | - Every **Extend** association identified in the *$\pi$-UseCase model* will be represented in the target model by a **Fork Node**.<br>- The **Service Activity** corresponding to the source **Use Case** of the extend association must be previous to the **Service Activity** corresponding to the target **Use Case** of the extend association.<br>- If the extend association has only one source **Use Case**, the fork will present the **Service Activity** as an alternative to another flow with no **Service Activity**. Later, both flows will meet;<br>- If the extend association has several source **Use Case**, the fork will present the different **Service Activities** as mutual alternatives to another flow with no **Service Activity**. Later, all these flows will meet; | Control Node,<br>Fork Node,<br>Service Activity |
| Include (Use Case) | - An **Include** association is found in the *$\pi$-UseCase model*, the **Service Activity** corresponding to the source must be subsequent to the **Service Activity** corresponding to the target **Use Case** of the include association;<br>- If the include association has several targets, the designer must decide the appropriate sequence for the different **Service Activities** corresponding to the target **Use Case** (which will obviously be previous to the **Service Activity** corresponding to the source **Use Case**). | Service Activity |
| Requirement | - A **Requirement** in the source model is transformed into **Service Activity** of the target model.<br>- The rules for a **Requirement** to be transformed into a service activity is analyzed after the transformations of included and extended use cases.<br>- A **Requirement** associated with only one **Use Case** in the source model is transformed in a. **Service Activity** and an **Action**, respectively.<br>- All **Use Cases** must be associated with a **Requirement**. | Service Activity |

type are transformed into an ASSERTION that does not consider value attributes information like *maxValue* and *minValue* (see figure 14). These ASSERTION information attributes are considered if the CONSTRAINT TYPE is a *Value* type. This is semi-automatic, because there is not enough information in a $\pi$-UseCase model to run a complete automatic transformation. If the CONSTRAINT TYPE is a *Value* type, the designer must specify the variable information, and its boundary values after the transformation. For this, the designer must consider the CONSTRAINT and USE CASE descriptions. By default, *value constraints* are transformed into *pre-conditions* and *business constraints* are transformed into *post-conditions*. This rule may be adjusted by the designer at the time of transformation.

A SERVICE ACTIVITY and the ACTIONS are generated from two different elements in the source model ($\pi$-UseCase), from a REQUIREMENT or a COMPOSITE USE CASE. Figures 36c



(a) From *Use Case* and *Constraint* to *Action* and *Contract*

(b) From *Constraint* to *Contract* (2)

(c) From *Composite Use Case* to *Service Activity*

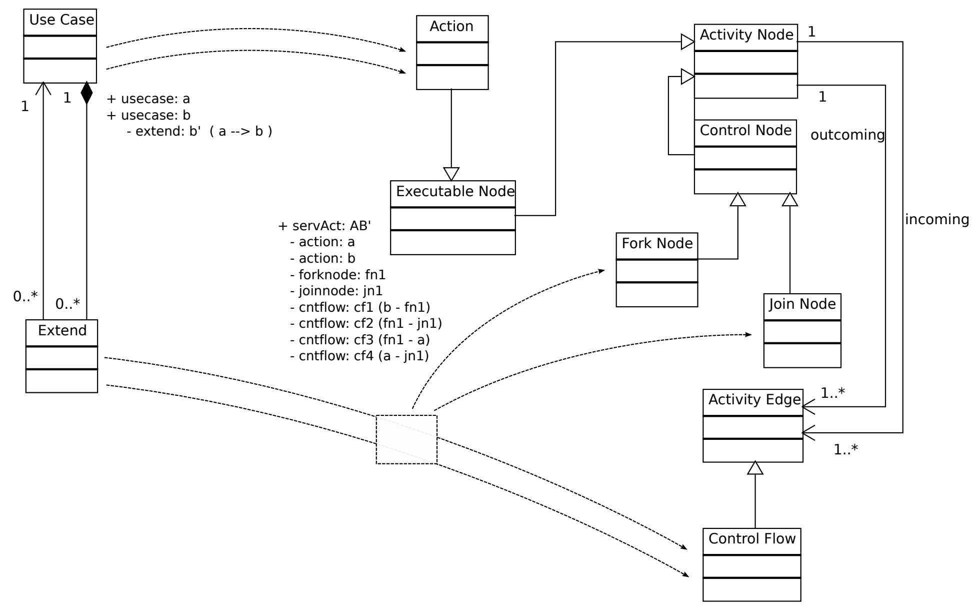
(d) From *Requirements* to *Service Activity*

Figure 36: $\pi$-UseCase2$\pi$-ServiceProcess Model Transformation Rules

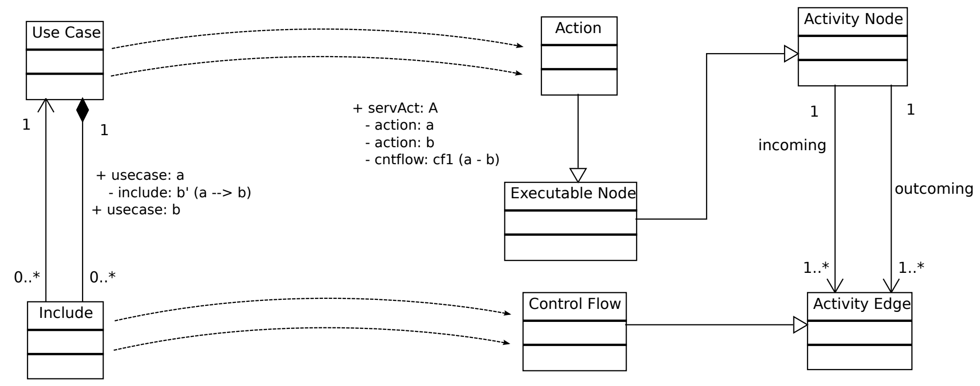
and 36d present the transformation schema for REQUIREMENT and COMPOSITE USE CASE concepts. Both elements are related with many USE CASES that are transformed in many ACTIONS. A USE CASE are related with, either, a REQUIREMENT and a COMPOSITE USE CASE. ACTIONS are related with a SERVICE ACTIVITY.

*Example:* For the execution of the "*download music*" use case is necessary process with the payment process. Thus, the set of USE CASES that include the "*download music*" process are transformed in ACTIONS, and a SERVICE ACTIVITY that aggregates all these ACTIONS is also generated. □

The transformations for EXTEND and INCLUDE dependence elements are not as simple as the previous transformations (figures 37a. The workflow generated for each *«extends»* relationship with just two use cases (figure 38a) is described by figure 38c. The generated workflow contains: one FORK NODE, one JOIN NODE, and four CONTROL FLOW elements, and also two ACTION elements, one for each USE CASE, if there are less than 2 extended USE CASE. When there is more than one *«extends»* relationship among different use cases (figures 38b and 38d), the transformation is proceed as: adding two more CONTROL FLOW for each new USE CASE, and one ACTION for each new extended USE CASE.



(a) From *Extend* to *Fork/Join Flow*



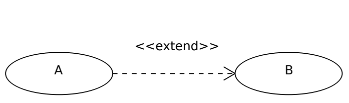
(b) From *Include* to *Activity Node Flow*

Figure 37: $\pi$-UseCase2$\pi$-ServiceProcess Model Transformation Rules (2)

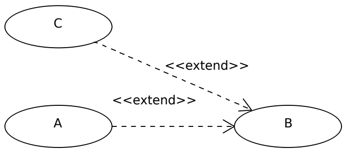
*Example:* Considering the scenario example, it is possible to apply the same rule for the "*publish music*" use case, which has two extended use cases, "*public twitter*" and "*public Facebook*", applying the illustration expressed in figures 38b and 38d. □

For the INCLUDE use case elements, the transformation is represented as a ACTION sequence, as shown in figure 37b. For each USE CASE element an ACTION element is generated. For a set of *n* USE CASES we generate *n-1* OBJECT FLOW elements. Each CONTROL FLOW links two ACTIONS.

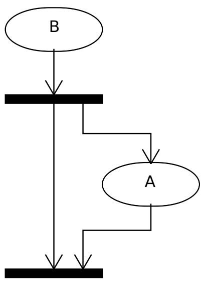
*Example:* Given the "*download music*" use case from the scenario example, it includes payment process to buy the music. It is represented as a include in the $\pi$-UseCase model and is transformed in a sequence flow in the $\pi$-ServiceProcess model (as presented in figure 39). □



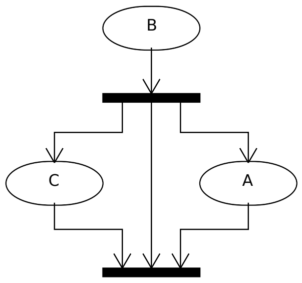
(a) Extended Use Case Example



(b) Extended Use Case Example (2)



(c) Equivalent Service Process Workflow



(d) Equivalent Service Process Workflow (2)

Figure 38: Extended Transformation Examples

All this transformation rules from $\pi$-*UseCase* model to $\pi$-*ServiceProcess* model were validated with the scenario example. All transformations described in this section are completely automatic, however information about values and its boundaries are inserted by the designer after the transformation process.

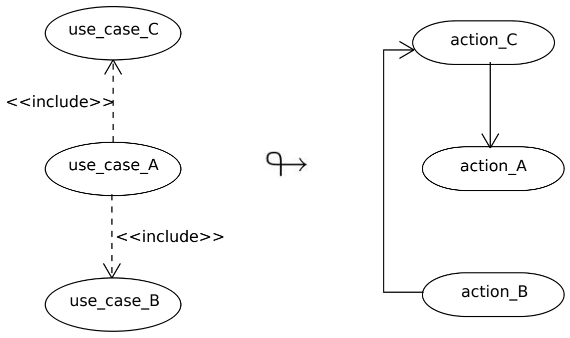


Figure 39: Include Transformation Example

### 3.4.2 From $\pi$-ServiceProcess to $\pi$-ServiceComposition

The PIM to PIM transformation process from $\pi$-*ServiceProcess* to $\pi$-*ServiceComposition* model will refine further the application design. The main goal of this transformation is to group all CONTRACTS and ACTIONS in POLICIES and SERVICE ACTIVITIES, respectively.

All ACTIONS entities in the source model will be transformed into an ACTION in the target model, and every SERVICE ACTIVITY in the source model will be transformed into a SERVICE ACTIVITY in the target model. This happens because $\pi$-*ServiceComposition* is an extension of $\pi$-*ServiceProcess*.

For the non-functional specifications, the CONTRACT entity with the same NON-FUNCTIONAL REQUIREMENT in the source model ($\pi$-*ServiceProcess* model) will be transformed into a POLICY in the target model ($\pi$-ServiceComposition model). Each ASSERTION will be transformed into a RULE:CONDITION attribute in the target model, but if the ASSERTION has a value type, the *name* and the attributes in the source model will be transformed into a VARIABLE in the target model. The other ASSERTION PROPERTY values remains unchanged.

The ASSERTION:APROPERTY attribute can have different transformations, depending of the following values:

- *POST-CONDITIONS* are transformed into *POST* in the $\pi$-ServiceComposition model;
- *PRECONDITIONS* are transformed into *PRE* in the $\pi$-ServiceComposition model;
- *TIMERESTRICTIONS* are transformed into *TIME* in the $\pi$-ServiceComposition model.

The EXCEPTIONAL BEHAVIOUR entities will be transformed into an ACTION in the $\pi$-*ServiceComposition* model, and every NON-FUNCTIONAL ATTRIBUTE associated to an element (CONTRACT and NON-FUNCTIONAL REQUIREMENT) in the $\pi$-ServiceProcess model becomes a NON-FUNCTIONAL ATTRIBUTE associated to the corresponding element (POLICY) in the $\pi$-*ServiceComposition* model.

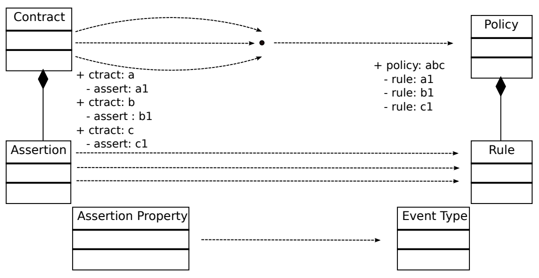
Table 6 describes the transformations between $\pi$-*ServiceProcess* and $\pi$-*ServiceComposition* meta-models.

Figure 40 presents the main transformation rules for this level. The main feature of this transformation is the generation of a POLICY from of a set of CONTRACTS. But, *what is the adopted criterion for this transformation?* The main criteria is that all CONTRACTS from the same NON-FUNCTIONAL REQUIREMENT will be transformed in one POLICY. Thus, we can have POLICIES on: *Safety, Performance, Reliability*, and so on.

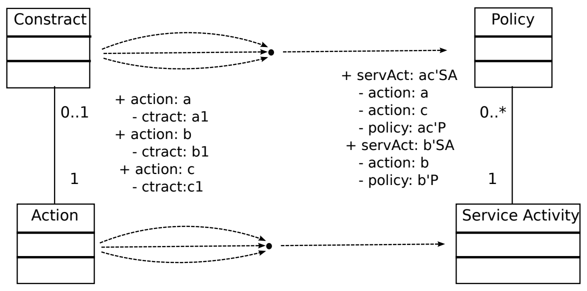
Each ASSERTION of aCONTRACT ( $\pi$-*UseCase* model) is transformed into a RULE in the $\pi$-*ServiceProcess* model. The set of CONSTRACTS generates a POLICY composed by those RULES. Thus, the set of RULES make up a single POLICY. It is important to highlight that only the CONTRACTS of the same NON-FUNCTIONAL REQUIREMENT is transformed into the same

Table 6: Transformation Rules: From $\pi$-ServiceProcess to $\pi$-ServiceComposition

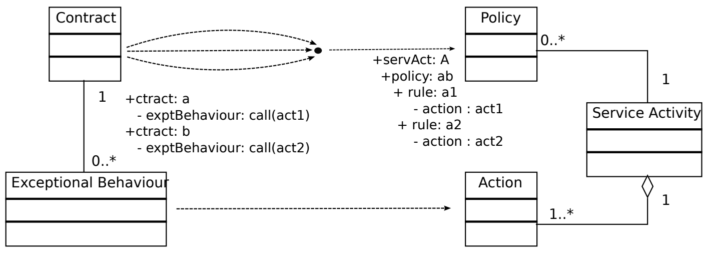
| Source<br>$\pi$-ServiceProcess meta-model | Mapping Rules | Target<br>$\pi$-ServiceComposition<br>meta-model |
|---|---|---|
| Action | All **Actions** in the source model will be transformed in a **Action** in the target model. | Action |
| Service Activity | - All **Service Activity** in the source model will be transformed in a **Service Activity** in the target model. | Service Activity, |
| Contract,<br>Non-Functional Attribute | All **Contract** with the same **Non-Functional Requirement**, in the source model, will be transformed in a **Policy** in the target model. | Policy |
| Assertion | Each **Assertion** will be transformed in a **Rule:condition** attribute in the target model. | Rule:condition |
| Assertion:name,<br>Assertion:type | If the **Assertion** has the type equal to *VALUE*, the *name* and the *type* attribute in the source model will be transformed in a **Variable** in the target model. | Variable |
| Exceptional Behaviour | An **Exceptional Behaviour** and its attributes in the source model will be transformed in a **Rule:action** in the target model. | Rule:action |
| Assertion:aProperty | - Depending on the **assertion Property** type in the source model, it will be transformed in a **Rule:event** in the target model. The transformation rules are:<br>● *PRECONDITION* type is transformed into *PRE* in the target model;<br>● *POST-CONDITION* type is transformed into *POST* in the target model;<br>● *TIMERESTRICTION* type is transformed into *TIME* in the target model. | Rule:Event,<br>Event Type |
| Non-Functional<br>Attribute | Every **Non-Functional Attribute** associated to an element (**Contract** and **Non-Functional Requirements**) in the source model becomes a **Non-Functional Attribute** associated to the corresponding element (**Policy**) in the target model. | Non-Functional<br>Attribute |



(a) From *Contract* and *Assertion* to *Policy* and *Rule*



(b) From *Action* to *Service Activity* (2)



(c) From *Exceptional Behaviour* to *Action*

Figure 40: $\pi$-ServiceProcess2$\pi$-ServiceComposition Model Transformation Rules

POLICY. Different NFRs' CONTRACTS, are transformed into different POLICIES. Figure 40a show how this transformation is made.

Similar to the transformation between CONTRACT and POLICY, a set of ACTIONS is transformed into a single SERVICE ACTIVITY entity. ACTIONS are related with BUSINESS COLLABORATOR, and all information related to a BUSINESS COLLABORATOR are transformed from PACKAGES information ($\pi$-UseCase model). All PACKAGES are transformed into BUSINESS COLLABORATOR. Figure 40b presents the transformation between ACTION and SER-

VICE ACTIVITY elements.

An exceptional behaviour is triggered by a failure in the verification of the condition of a rule. An EXCEPTIONAL BEHAVIOUR is performed to preserve the state of the application. Figure 40c presents how an EXCEPTIONAL BEHAVIOUR that can be transformed into an ACTION call.

As the *$\pi$-ServiceComposition* model refines the *$\pi$-ServiceProcess* concepts at PIM level, a service previously defined as actions (source model) is refined as composition of those actions (target model) that are necessary to represent a business service, identifying who are the partners involved in the realization (BUSINESS COLLABORATORS). In addition, $\pi$SOD-M defines a platform specific model based on web services composition. This model is explicitly indicates those actions which are (or will be, if not yet implemented) supported by web services.

*Example:* Considering the scenario example, the ACTION's *update music* CONTRACT is transformed is a POLICY with its RULES. All contract ASSERTIONS are transformed in RULE and its attributes, *e.g.* the login and password verification. The "*securityLoginPolicy*" is all set of RULES that were transformed from the ASSERTIONS in $\pi$-ServiceProcess model. The NON-FUNCTIONAL REQUIREMENT information will be used to the POLICY generation comes from the initial use case model. Also the BUSINESS COLLABORATOR *Facebook* and *Spotify* information came from PACKAGE $\pi$-UseCase entity element. All CONTRACTS of the same NON-FUNCTIONAL REQUIREMENT are composed in a POLICY. □

### 3.4.3 From $\pi$-ServiceComposition to $\pi$-PEWS

Table 7 describes the PIM to PSM transformations between $\pi$-ServiceComposition and $\pi$-PEWS meta-models. We propose two groups of rules: those that transform services composition elements of the $\pi$-ServiceComposition into $\pi$-PEWS meta-model elements; and those that transform rules grouped by policies into A-policy types.

A named action of the $\pi$-ServiceComposition represented by *Action* and *Action:name* is transformed to a named class OPERATION with a corresponding attribute name OPERATION:NAME. A named service activity represented by the elements *Service Activity* and *Service Activity:name* of the $\pi$-ServiceComposition, are transformed into a named operation of the $\pi$-PEWS represented by the elements COMPOSITE OPERATION and COMPOSITE OPERATION:NAME. When more than one action is called, according to the following composition patterns expressed using the operators *merge, decision, fork and join* in the $\pi$-ServiceComposition the corresponding transformations, according to the PEWS operators presented above, are:

- $op_1.op_2$ if no *Control Node* is specified
- $(op_1 \parallel op_2).op_3$ if control nodes of type *fork, join* are combined
- $(op_1 + op_2).op_3$ if control nodes of type *decision, merge* are combined

The *A-policies* defined for the elements of the $\pi$-ServiceComposition are transformed into A-POLICY classes, named according to the names expressed in the source model. The transformation of the rules expressed in the $\pi$-ServiceComposition is guided by the event types associated to these rules. The variables associated to an *A-policy* expressed in the $\pi$-ServiceComposition as <*Variable:name, Variable:type*> are transformed into elements of type VARIABLE with attributes NAME and TYPE directly specified from the elements *Variable:name* and *Variable:type* of the $\pi$-ServiceComposition model.

Table 7: Transformation Rules: From $\pi$-ServiceComposition to $\pi$-PEWS

| Source<br>$\pi$-ServiceComposition meta-model | Mapping Rules | Target<br>$\pi$-PEWS meta-model |
|---|---|---|
| Action | - An *Action* in the source model corresponding to an external **Business Collaborator** is mapped to an Operation in target model.<br>- The **Action:name** in the source model is transformed into **Operation:name** in the target model. | Operation:alias |
| Service Activity | - The **Service Activity** in the source model is mapped to a **Composite Operation** in target model when more than one **Actions** are called.<br>- If **Composite Operation** is generated for a given *Service Activity* then the **Service Activity:name** in the source model is mapped to **Composition Operation:name** in the target model. | Type Operation,<br>Composition Operation |
| Control Nodes | - The **Control Node** in the source model is mapped to a **Operator** in target model. According to the type of **Control Node** (merge, decision, join, fork) the expression of the **Composite Operation** is:<br>● Sequence if no **Control Node** is specified;<br>● Parallel - Sequence for a **Control Nodes** pattern fork - join;<br>● Choice - Sequence for a **Control Node** pattern decision - merge | Operator |
| Business Collaborator | A **Business Collaborator:isExternal** in the source model generates a **Namespaces** in the target model | Namespace |
| Rule:event | The **Rules**´ attribute event in the source model is transformed into an **Event:type** of the target model. In this case attribute is mapped to an entity with an attribute. The **Event Type** of a **Rule** in the target model is determined by the Rule type:<br>● **Event Type** of a *Precondition* **Rule** is *ActivityPrepared*;<br>● **Event Type** of a *Postcondition* **Rule** is *TermActivity*;<br>● **Event Type** of a *TimeRestriction* **Rule** is *Activity*; | Event Type,<br>Event |
| Rule:condition | The **Rules**´ attribute condition in the source model is transformed into a **Condition:expression** in the target model. In this case, an attribute is mapped into an entity with an attribute | Condition |
| Rule:action | The **Rule:action** in the source model is transformed in an **Action:act** in the target model. The attribute action is mapped to an entity with an attribute. In the target model an action is executed according to the rule condition value (true/false). | Action |
| Policy | - Every **Policy** associated to an element (**Business Collaborator, Service, Activity, Action**) in the source model becomes an **APolicy** associated to the corresponding element in the target model.<br>- The name attribute of a **Policy** in the source model becomes an **Apolicy:name** of the target model. | APolicy |
| Variable | Every **Variable**, and its attributes, associated to a **Policy** in the source model becomes a **Variable** associated to an **APolicy** in the target model. The variables can be used in an **APolicy**´ Condition of the target model. | Variable |
| Rule:event | For a **Rule** in the source model, depending on the **Event Type**, the corresponding transformation in the target model is: **Precondition, Postcondition** or **TimeRestriction Rule** | Precondition<br>Postcondition<br>TimeRestriction<br>Rule |

As shown in Table 7, for an event of type *Pre* the corresponding transformed rule is of type PRECONDITION; for an event of type *Post* the corresponding transformed rule is of type POSTCONDITION; finally, for an event of type *TimeRestriction* the corresponding transformed rule is of type TIME. The condition expression of a rule in the $\pi$-ServiceComposition (*Rule:condition*) is transformed into a class *Condition:expression* where the attributes of the expression are transformed into elements of type ATTRIBUTE.

Figures 32 and 33 present the $\pi$-*PEWS* specification resulting from the $\pi$-*ServiceComposition* model transformation of our scenario example.

## 3.5 Conclusions

This chapter presented $\pi$-SOD-M for specifying and designing reliable service based applications, using a MDA approach. As one of the main aims of MDA is to separate design from architecture and technologies, the $\pi$SOD-M's models describe the system behaviour and its restrictions, without considering definition of a standard architecture.

Non-functional constraints are related to business rules associated to the general semantics of the application. In the case of service-based applications, this type of applications also concern the use constraints imposed by the services. We worked on the definition of a method for explicitly expressing such properties in the early stages of the specification of services based applications. Having such business rules expressed and then translated and associated to the services' composition can help to ensure that the resulting application is compliant to the user requirements and also to the characteristics of the services it uses.

Designing and programming non-functional properties is not an easy task, so we are defining a set of predefined *Policy* types with the associated use rules for guiding the programmer when she associates them to a concrete application. *Policy* types that can also serve as patterns for programming or specializing the way non-functional constraints are programmed. We model and associate policies to service-based applications that represent both systems' cross-cutting aspects and use constraints stemming from the services used for implementing them.

An advantage of our methodology is the use of high-level models, which by means of model transformations, helps software developers to make the most of the knowledge for specifying and developing business services.

# 4 πSOD-M Environment

"*The computer was born to solve problems that did not exist before.*"

Bill Gates

This chapter describes the $\pi$SOD-M environment for developing reliable service based systems using the methodology we propose. The $\pi$SOD-M environment was developed to support the $\pi$SOD-M methodology. The $\pi$SOD-M based process consists in generating an application starting from a $\pi$-*UseCase* model and then transforming it into series of models at the PIM and PSM levels before finally generating the code that implements the application.

$\pi$SOD-M environment is built on the top of Eclipse framework (http://www.eclipse.org), which is a framework to build Integrated Development Environments (IDEs). We also use the Eclipse Modelling Framework (EMF), a meta-modelling framework that was devised to be extended and provides with the utilities needed to define, edit and handle (meta)-models. To automate the transformation models we use the ATL Language[1] [49], a model transformation language framed in Eclipse. Another language used for model transformation is Acceleo [72], a model to text engine. $\pi$SOD-M uses this environment to generate service composition specification code in $\pi$-*PEWS*.

The remainder of the chapter is organized as follows. Section 4.1 describes the environment architecture, including the definition of meta-models; and, the implementation of the model transformation process and the code generation engine. In section 4.2 describes how to install and use the environment. Section 4.3 describes how to extend the environment, for adding new components for generation of system specification in different languages. Section 4.4 concludes the chapter.

[1]ATL uses EMF to handle models (that is to serialize, to navigate and to modify them). Once the ATL transformation is executed, the ATL engine matches the source pattern and the source model and instantiates each pattern into the target model.

## 4.1 General Architecture

Figure 41 presents the general architecture for the $\pi$SOD-M environment and details how components interact with each other. Each model has an editor, and transformation tools that support the application development.

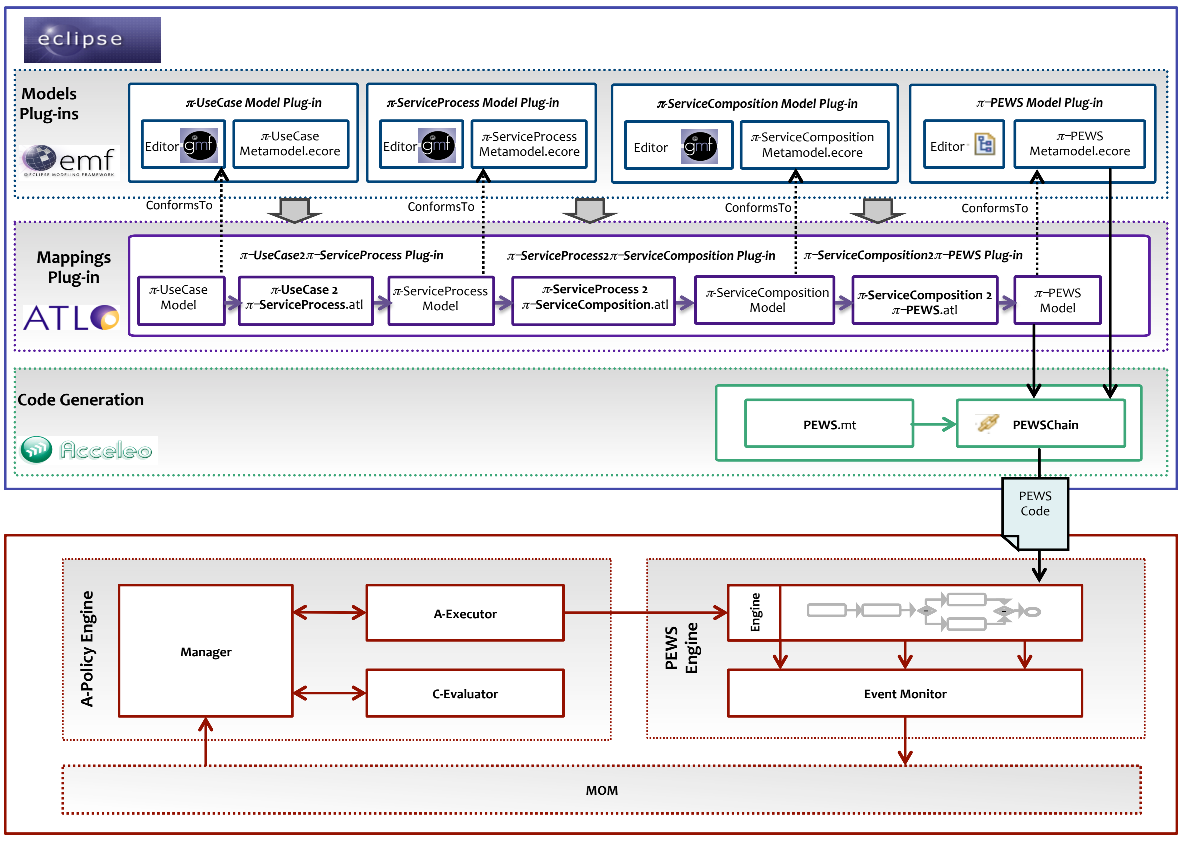

eclipse
Models Plug-ins
emf
π-UseCase Model Plug-in
Editor
π-UseCase Metamodel.ecore
π-ServiceProcess Model Plug-in
Editor
π-ServiceProcess Metamodel.ecore
π-ServiceComposition Model Plug-in
Editor
π-ServiceComposition Metamodel.ecore
π-PEWS Model Plug-in
Editor
π-PEWS Metamodel.ecore
ConformsTo
ConformsTo
ConformsTo
ConformsTo
Mappings Plug-in
ATL
π-UseCase2π-ServiceProcess Plug-in
π-ServiceProcess2π-ServiceComposition Plug-in
π-ServiceComposition2π-PEWS Plug-in
π-UseCase Model
π-UseCase 2 π-ServiceProcess.atl
π-ServiceProcess Model
π-ServiceProcess 2 π-ServiceComposition.atl
π-ServiceComposition Model
π-ServiceComposition 2 π-PEWS.atl
π-PEWS Model
Code Generation
Acceleo
PEWS.mt
PEWSChain
PEWS Code
A-Policy Engine
Manager
A-Executor
C-Evaluator
PEWS Engine
Engine
Event Monitor
MOM

Figure 41: $\pi$SOD-M Development Environment.

The $\pi$SOD-M environment architecture is organized in three layers: *(i) Meta-model level, (ii) PIM-to-PIM Transformations level level, (iii) PIM-to-PSM Transformations level level* and *(iv) Code generation level*. Each layer comprises a set of components that together support the $\pi$SOD-M environment. This figure presents how those components are implemented by our tool: (i) the *Meta-model component* (figure 42) represents the *Model plugins module* of figure 41 (together with all methodology meta-models); (ii) the *PIM-to-PIM Transformations and PIM-to-PSM Transformations components* (figure 42) represent the *Mapping plugins module* of figure 41; (iii) the *Code Transformation* component of the figure 42 is implemented by the *Code generation module* of figure 41; and (iv) the *Execution engine* is represented by the lower tier of figure 41.

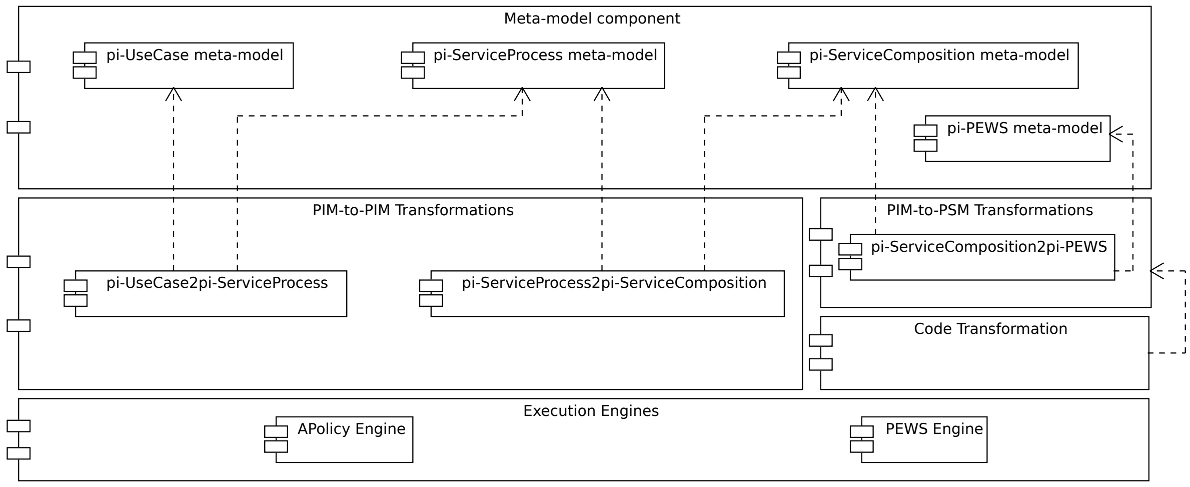

Meta-model component
pi-UseCase meta-model
pi-ServiceProcess meta-model
pi-ServiceComposition meta-model
pi-PEWS meta-model
PIM-to-PIM Transformations
pi-UseCase2pi-ServiceProcess
pi-ServiceProcess2pi-ServiceComposition
PIM-to-PSM Transformations
pi-ServiceComposition2pi-PEWS
Code Transformation
Execution Engines
APolicy Engine
PEWS Engine

Figure 42: Environment Components

The *Models plugin module* comprises the components that describe the $\pi$SOD-M meta-models, and how their models must be created. There are four meta-model components. All components of the *Mapping plugin module* depend of the definitions made in *Models plugin*.

When a model transformation is made, the models must comply with their respective meta-model. The process is executed as follows: every time a transformation is performed, a consistency check of both, source and target models is performed. After all transformation are made, the PSM model is translated into code of a particular platform, in the case of $\pi$SOD-M, $\pi$-*PEWS* is the chosen platform. The transformation of PSM model in code is the last stage of transformation. The component of the *Code generation module* depend of the PSM generated by the last model transformation. Finally, the *Execution Engine* component performs the execution of the service-based specification code.

### 4.1.1 Ecore Meta-models (*Models Plugin Module*)

The implementation of each meta-model is defined in *Ecore*[2] files, they are: $\pi$-*UseCase,* $\pi$-*ServiceProcess,* $\pi$-*ServiceComposition* and $\pi$-*PEWS*. All meta-models have a related *genmodel*[3] definition. The plugin editor can be created from the *genmodel* definition, so that models can be specified.

Using these set of tools (model editors) it is possible to create models at different $\pi$SOD-M

[2]The *Models Plugin Module* is a set of Ecore files that represents all $\pi$SOD-M meta-models. These meta-models are the sources for the development of each of the methodology's model. All models designed for an application must obey their meta-model specification. This module is composed by all the proposed methodology's models in EMF (*.ecore extension*).

[3]A *.genmodel* is a intermediary file format used to produce the syntax editor for each meta-model.

levels. There are editors for all $\pi$SOD-M models: *$\pi$-UseCase editor, $\pi$-ServiceProcess editor, $\pi$-ServiceComposition* editor and *$\pi$-PEWS* editor.

Although there are editors for each methodology model, it is still necessary components to perform transformations among them. Thus, the model specification process can be made in all methodology levels. However it can be made only at the highest level, *i.e.* *$\pi$-UseCase* model, and then execute automatic transformation to generate lowest level models.

### 4.1.2 Model Transformation (*Mapping Plugin Module*)

The *Model transformation level* has a set of components for processing and transforming models. The model transformation components are based on the source models for generating the equivalent target model. For example, from a *$\pi$-UseCase* model is generated the *$\pi$-ServiceProcess* model, from a *$\pi$-ServiceProcess* model is generated the *$\pi$-ServiceComposition* model, and finally, from a *$\pi$-ServiceComposition* model is generated a *$\pi$-PEWS* model. After the model generation the designer can perform refinements. Refinement can improve and adjust elements that require a more detailed description at this modeling level.

$\pi$SOD-M model transformation process is based on the rules described in chapter 3. The implementation process requires additional, more specific information to be taken into account. We had to consider the representation of our concepts in the Eclipse and ATL environments for MDA-based application development, *e.g.*, aspects of plugin generation; design model properties; compliance of the designed meta-model with the generated model; the specific model implementation; and etc.

We used the Eclipse Modeling Framework (EMF) for implementing the meta-models *$\pi$-UseCase, $\pi$-ServiceProcess, $\pi$-ServiceComposition* and *$\pi$-PEWS*. The ATL language was used for developing the mapping rules for transformation of models (*$\pi$-UseCase2$\pi$-ServiceProcess*, *$\pi$-ServiceProcess2$\pi$-ServiceComposition* and *$\pi$-ServiceComposition2$\pi$-PEWS* plug-ins). This plugin takes as input a *$\pi$-PEWS* model implementing a specific service composition and it generates the code to be executed by the *Policy* based *Execution Engine* (figure 42).

Figures 43 and 44 present a general view of the $\pi$SOD-M models transformation, showing the set of plug-ins developed to implement it. Figures 43 presents the model-to-model transformations, while figure 44 shows the model-to-text transformation schema. The environment implements the abstract architecture shown in figure 41. This environment consists of plug-ins implementing the *$\pi$-UseCase, $\pi$-ServiceProcess, $\pi$-ServiceComposition* and *$\pi$-PEWS* meta-models used for defining models; and ATL rules for transforming PIM and PSM mod-

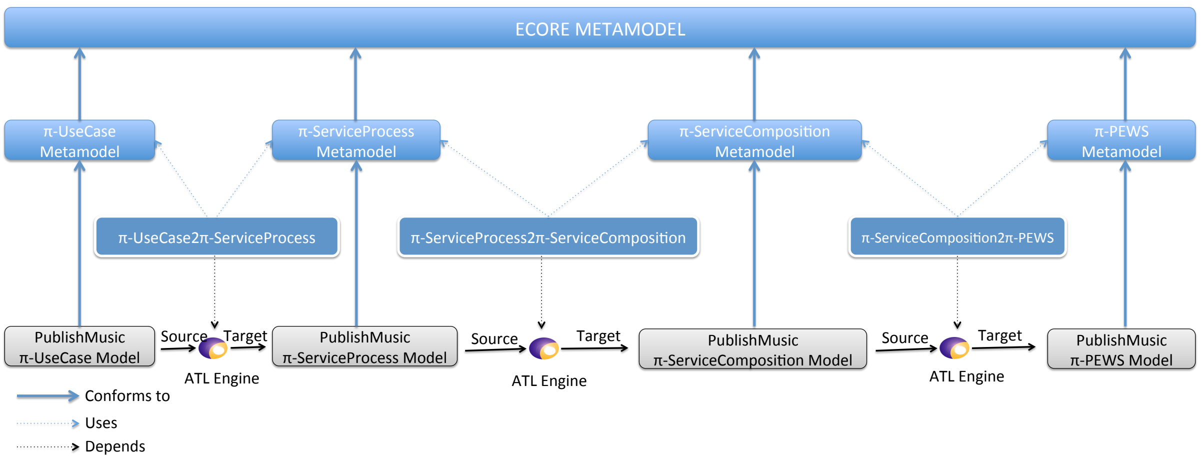

ECORE METAMODEL
π-UseCase Metamodel
π-ServiceProcess Metamodel
π-ServiceComposition Metamodel
π-PEWS Metamodel
π-UseCase2π-ServiceProcess
π-ServiceProcess2π-ServiceComposition
π-ServiceComposition2π-PEWS
PublishMusic π-UseCase Model
Source
Target
ATL Engine
PublishMusic π-ServiceProcess Model
Source
Target
ATL Engine
PublishMusic π-ServiceComposition Model
Source
Target
ATL Engine
PublishMusic π-PEWS Model
Conforms to
Uses
Depends

Figure 43: ATL Model to Model Transformation in $\pi$SOD-M.

els (model to model transformation) and finally generating code (model to text transformation) with Acceleo.

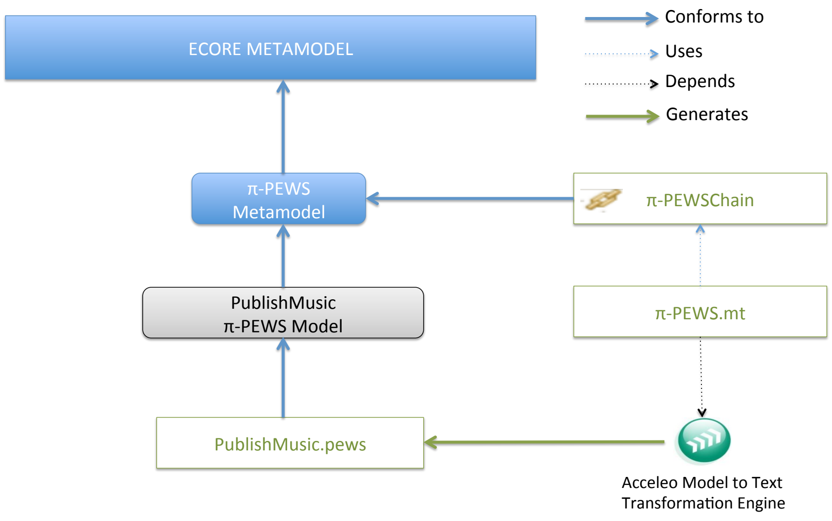
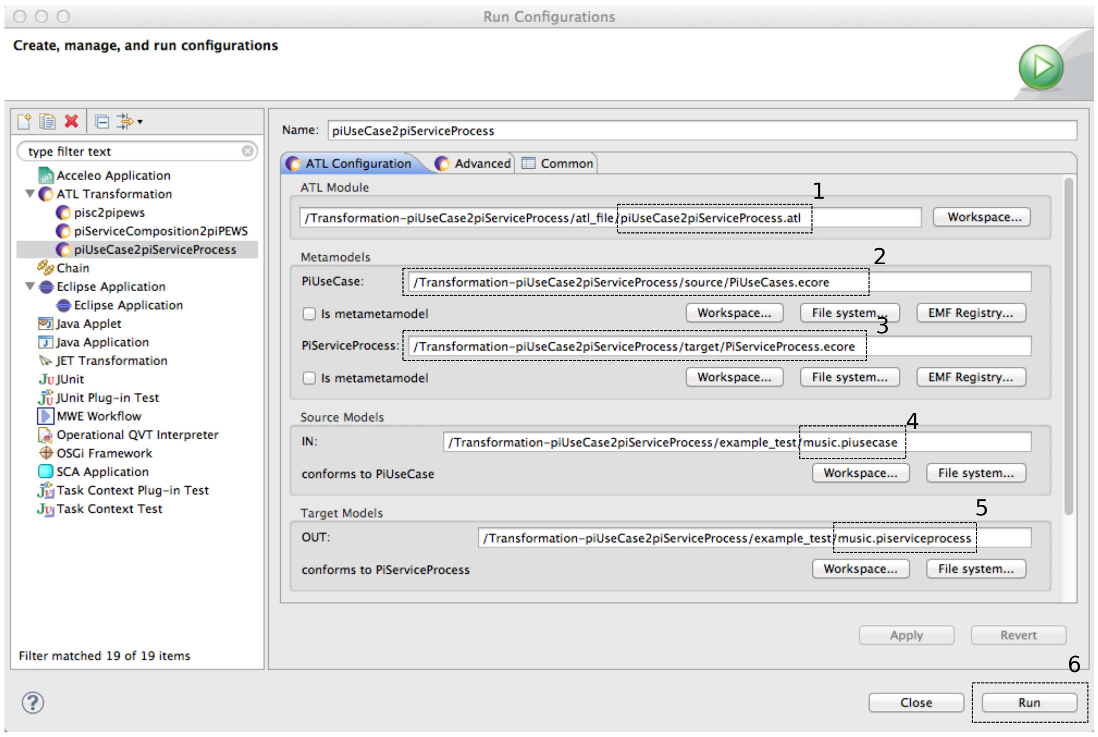


Figure 44: Acceleo Model to Text Transformation in $\pi$SOD-M.

In order to proceed with a model transformation it is necessary to configure the transformation environment. Figure 45 presents a standard screen for configuring each transformation (in this case for the "to publish music" scenario). From any source model, for example, *$\pi$-UseCase, $\pi$-ServiceProcess* or *$\pi$-ServiceComposition*, the system can perform the automatic transformation. Using the transformation tool of figure 45 requires from the user: (1) to indicate the ATL transformation rules file; (2) to choose the source meta-model reference (.ecore file); (3) to choose the target meta-model reference (.ecore file); (4) to choose the source model that will be transformed (*e.g. music.piusecase* file); (5) to choose the target model that will be generated (*e.g. music.piserviceprocess* file); and finally, (6) to run the tool. The same must be performed for all model-to-model transformations. For model-to-text transformation, the rule file to be chosen should be the Acceleo file.



Name: piUseCase2piServiceProcess

ATL Module
/Transformation-piUseCase2piServiceProcess/atl_file/piUseCase2piServiceProcess.atl

Metamodels
PiUseCase: /Transformation-piUseCase2piServiceProcess/source/PiUseCases.ecore
Is metametamodel
PiServiceProcess: /Transformation-piUseCase2piServiceProcess/target/PiServiceProcess.ecore
Is metametamodel

Source Models
IN: /Transformation-piUseCase2piServiceProcess/example_test/music.piusecase
conforms to PiUseCase

Target Models
OUT: /Transformation-piUseCase2piServiceProcess/example_test/music.piserviceprocess
conforms to PiServiceProcess

Figure 45: ATL Configuration for $\pi$SOD-M Transformation.

As an example, figure 45 shows the configuration for the transformation from $\pi$-*UseCase* (source) to $\pi$-*ServiceProcess* (target). Notice that, in this case, the reference meta-models must be the same. For further transformations, this process must follow the same sequence, changing only the models and reference meta-models.

#### 4.1.2.1 $\pi$-*UseCase2$\pi$ServiceProcess* Transformation Rules

The transformation rules describe how models are transformed. As a general rule, the automatic transformation of models favors a faster development of applications. In most cased, the environment or the designer should verify if the transformations are valid or not. In the $\pi$SOD-M environment, the consistency of the transformations must be performed by the designer. If a problem on the transformation is identified, the target model can be modified manually.

In ATL, there exist two different kinds of rules that correspond to the two different programming modes provided by ATL (e.g. declarative and imperative programming). The *matched*

Listing 4.1: ATL Example Rule

```
1  rule ruleName{
2          from
3                  var_sourceName: SourceModel!Entity
4          to
5                  var_targetName: TargetModel!Entity(
6                          atribute_1 <- var_sourceName.atribute_a,
7                          atribute_2 <- var_sourceName.atribute_b
8                  )
9  }
```

Listing 4.2: ATL - piServiceProcess2piServiceComposition : root Rule

```
1  rule root {
2        from
3                root: piServiceProcess!ServiceProcess
4        to
5                root_service: piServiceComposition!CompositionServiceModel(
6                        activities <- root.activity,
7                        edges <- root.edge,
8                        compositionPolices <- root.contract
9                )
10 }
```

*rules*[4] (declarative programming) and the *called rules*[5] (imperative programming) [49]. For the $\pi$SOD-M environment, we use both types of rules. Listing 4.1 shows a general example of an ATL *matched rule*. According to this, we will present the main ATL rules used to implement the model transformation. As described in listing **??**, a rule consists of a name (line 1), a source entity (*from* clause in lines 2-3), and one or more target entities (*to* clause in line 4-5). Each entity has a name, such as $var_sourceNameand$

*As the $\pi$-ServiceComposition model is a refinement of some concepts defined in $\pi$-ServiceProcess, such as* CONTRACT *and* ACTIVITY SERVICES, *most of the transformation between these models are through direct transformation, without restrictions. We present the ATL transformation (listing 4.2) of the root element that comprises the main elements of both models.*

*Listing 4.2 shows the transformation rule for the main elements from $\pi$-ServiceProcess to $\pi$-ServiceComposition model. The* ACTIVITIES *and its* EDGES, *and* CONTRACTS, *which are transformed into* POLICY *(lines 6-8). Other rules that describe the transformation between* ASSERTIONS *into policy* RULE *are other type of transformation. However, other rules describe*

[4]The matched rules constitute the core of an ATL declarative transformation since they make it possible to specify 1) for which kinds of source elements target elements must be generated, and 2) the way the generated target elements have to be initialized. A matched rule is identified by its name. It matches a given type of source model element, and generates one or more kinds of target model elements. The rule specifies the way generated target model elements must be initialized from each matched source model element[49].

[5]The called rules provide ATL developers with convenient imperative programming facilities. Called rules can be seen as a particular type of helpers: they have to be explicitly called to be executed and they can accept parameters. However, as opposed to helpers, called rules can generate target model elements as matched rules do. A called rule has to be called from an imperative code section, either from a match rule or another called rule[49].

Listing 4.3: ATL - piServiceComposition2piPEWS : root Rule

```
1  rule root{
2          from sCM : pisc!CompositionServiceModel
3          to
4                  path: pipews!Path (),
5                  pews : pipews!PEWSCTSpec (
6                          name <- 'newModelPEWSpecName',
7                          has <- path,
8                          contains <- sCM.partition,
9                          defines <-        thisModule.policies
10                 )
11  }
```

*that all* CONTRACTS *must belong to a specific* NON-FUNCTIONAL REQUIREMENT*, for example, all contracts for the performance restrictions, will be grouped into a single performance* POLICY. *This listing is an ATL example for π-ServiceProcess model transformation into to π-ServiceComposition model.*

### 4.1.2.2 *π-ServiceComposition2π-PEWS* Transformation Rules

*The transformation π-ServiceComposition2π-PEWS is unique among PIM and PSM levels, however there is no difference in the rules description in ATL, since all rules are defined in terms of meta-models rather than specific models. Thus, PIM-to-PSM transformation rules in ATL follow the same syntax and semantics of the PIM-to-PIM transformation rules.*

*The model to be generated in this transformation, namely π-PEWS model, will be used as input for the code generation component, so that the application code specification is generated. The listings 4.3 and 4.4 present two transformation rules in ATL. The first describes the transformations of the main elements for the description of a service composition, the main path, the name of the specification, the services and policies (lines 6 - 9), while listing 4.4 describes a policy* RULE *and its related concepts, such as* ACTION, EVENT *and* CONDITION. *The ATL rule has a constraint to be checked (line 4), what kind of* RULE *is being translated for the specific language, because depending on the type, the transformation will be change. The* RULE *element (line 15) consists of all the properties necessary to create a* PRE-CONDITION*, such as* ACTION, EVENT *and* CONDITION *(lines 16-18), and to which* POLICY *the* PRE-CONDITION *is related (line 19).*

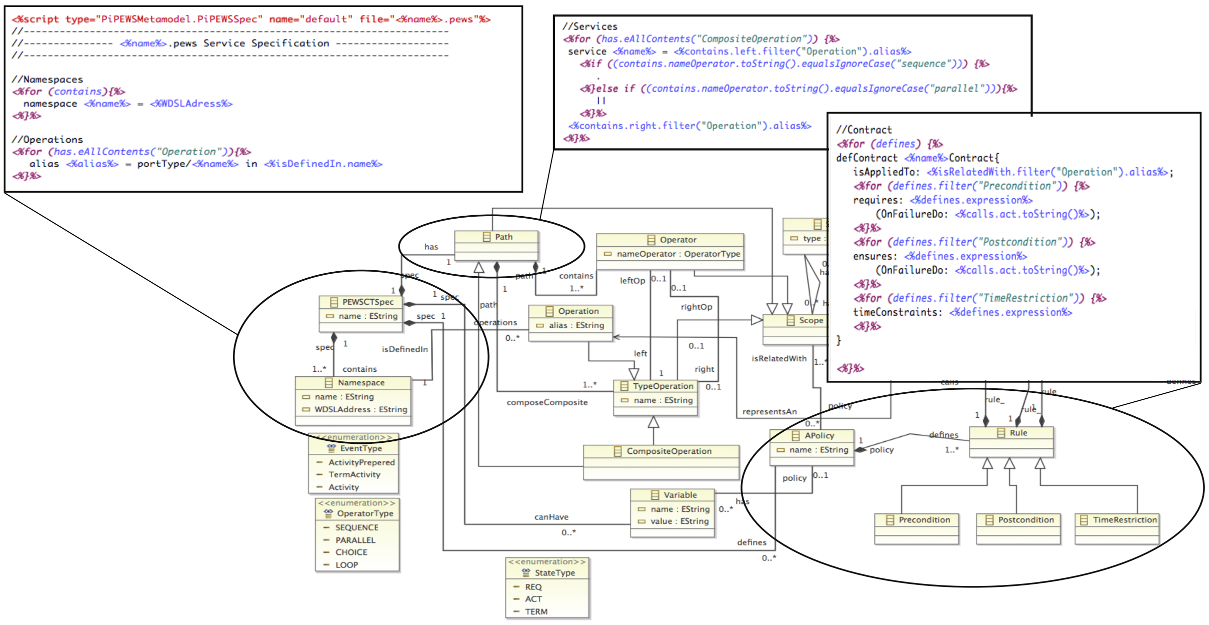
## 4.1.3 Code Generation (*Code Generation Module*)

The πSOD-M architecture's code generation component (*Code generation level*) is a *π-PEWS* specification generator. The code is produced from a *π-PEWS* model, after it be generated by a model transformation from *π-ServiceComposition* model. This component was

Listing 4.4: ATL - piServiceComposition2piPEWS : Pre-condition Rule

```
rule rulePre{
        from
                r: pisc!Rule (
                        r.event = #PRE)
        to
                rAct: pipews!Action(
                        act <- r.action
                ),
                rEvt: pipews!Event(
                        type <- #ActivityPrepered
                ),
                rCond: pipews!Condition(
                        expression <- r.condition
                ),
                rRule: pipews!Precondition(
                        calls <- rAct,
                        defines <- rCond,
                        hasSome <- rEvt,
                        policy <- r.policy
                )
 }
```

implemented using Acceleo [72]. Figure 47 presents a sequence diagram describing the model transformation process and how the designer interacts with the environment to specify each $\pi$SOD-M model until the specification code is generated.



```
<%script type="PiPEWSMetamodel.PiPEWSSpec" name="default" file="<%name%>.pews"%>
//----------------------------------------------------------------------
//-------------- <%name%>.pews Service Specification ------------------
//----------------------------------------------------------------------

//Namespaces
<%for (contains){%>
 namespace <%name%> = <%WDSLAdress%>
<%}%>

//Operations
<%for (has.eAllContents("Operation")){%>
  alias <%alias%> = portType/<%name%> in <%isDefinedIn.name%>
<%}%>
```

```
//Services
<%for (has.eAllContents("CompositeOperation")) {%>
 service <%name%> = <%contains.left.filter("Operation").alias%>
   <%if ((contains.nameOperator.toString().equalsIgnoreCase("sequence"))) {%>
     .
   <%}else if ((contains.nameOperator.toString().equalsIgnoreCase("parallel"))){%>
     ||
   <%}%>
 <%contains.right.filter("Operation").alias%>
<%}%>
```

```
//Contract
<%for (defines) {%>
defContract <%name%>Contract{
   isAppliedTo: <%isRelatedWith.filter("Operation").alias%>;
   <%for (defines.filter("Precondition")) {%>
   requires: <%defines.expression%>
      (OnFailureDo: <%calls.act.toString()%>);
   <%}%>
   <%for (defines.filter("Postcondition")) {%>
   ensures: <%defines.expression%>
      (OnFailureDo: <%calls.act.toString()%>);
   <%}%>
   <%for (defines.filter("TimeRestriction")) {%>
   timeConstraints: <%defines.expression%>
   <%}%>
}

<%}%>
```

Figure 46: Acceleo Specification for $\pi$-*PEWS* Code Generation.

Figure 46 presents the $\pi$-*PEWS* meta-model developed using EMF and some pieces of Acceleo specification. This figure shows the specification for the *Namescape, Operation, Service* and *Contract* code generation for $\pi$-*PEWS*. After the code transformation process, a *.pews* file is created. The listing 4.5, 4.6 and 4.7 present parts of Acceleo code for the $\pi$-*PEWS* code generation. The code are the same presented in figure 46, which presents the relation between the meta-model concepts and the Acceleo code. The code generation follow the language syntax described in appendix C.

Listing 4.5: Acceleo - Namespace and Operation Code Specification

```
1  <%script type="PiPEWSMetamodel.PEWSSpec" name="default"
2   file="<%name%>.pews"%>
3  //——————————————————————————————————————————————
4  //———————————— <%name%>.pews Service Specification ————————————
5  //——————————————————————————————————————————————

7   //Namespaces
8  <%for (contains){%>
9     namespace <%name%> = <%WDSLAdress%>
10  <%}%>

12   //Operations
13  <%for (has.eAllContents("Operation")){%>
14     alias <%alias%> = portType/<%name%> in <%isDefinedIn.name%>
15  <%}%>
```

Listing 4.6: Acceleo - Service Code Specification

```
1    //Services
2   <%for (has.eAllContents("CompositeOperation")) {%>
3      service <%name%> = <%contains.left.filter("Operation").alias%>
4     <%if ((contains.nameOperator.toString().equalsIgnoreCase("sequence"))) {%>
5          .
6     <%}else if((contains.nameOperator.toString().equalsIgnoreCase("parallel"))){%>
7          ||
8     <%}%>
9    <%contains.right.filter("Operation").alias%>
10  <%}%>
```

In listing 4.5 (lines 1-2) references the meta-model, the root element (*PEWSSpec*), and the name of the generated file (*<%name%>.pews*). Lines 3-5 presents the name of the service specification. Lines 7-10 describes a *for* integration in the *contains* relationship between *PEWSSpec* and *Namespace*, and lines 12-15 all operations defined in each *Namespace* is generated. This operations came from the *isDefinedIn* relationship between *Namespace* and *Operation* entities.

Listing 4.6 presents the Acceleo specification for the $\pi$-*PEWS* model transformation into code. A service is an alias for one or more operations. Listing 4.7 specifies the contract generation, using the *defines* relationship between *APolicy* and *Rule*. Each contract have a set of rules for the specification.

Listing 4.7: Acceleo - PEWS Contract Code Specification

```
1   //Contract
2  <%for (defines) {%>
3   defContract <%name%>Contract{
4           isAppliedTo: <%isRelatedWith.filter("Operation").alias%>;
5          <%for (defines.filter("Precondition")) {%>
6           requires: <%defines.expression%>
7                   (OnFailureDo: <%calls.act.toString()%>);
8          <%}%>
9          <%for (defines.filter("Postcondition")) {%>
10          ensures: <%defines.expression%>
11                  (OnFailureDo: <%calls.act.toString()%>);
12         <%}%>
13         <%for (defines.filter("TimeRestriction")) {%>
14          timeConstraints: <%defines.expression%>
15         <%}%>
16  }

 <%}%>
```

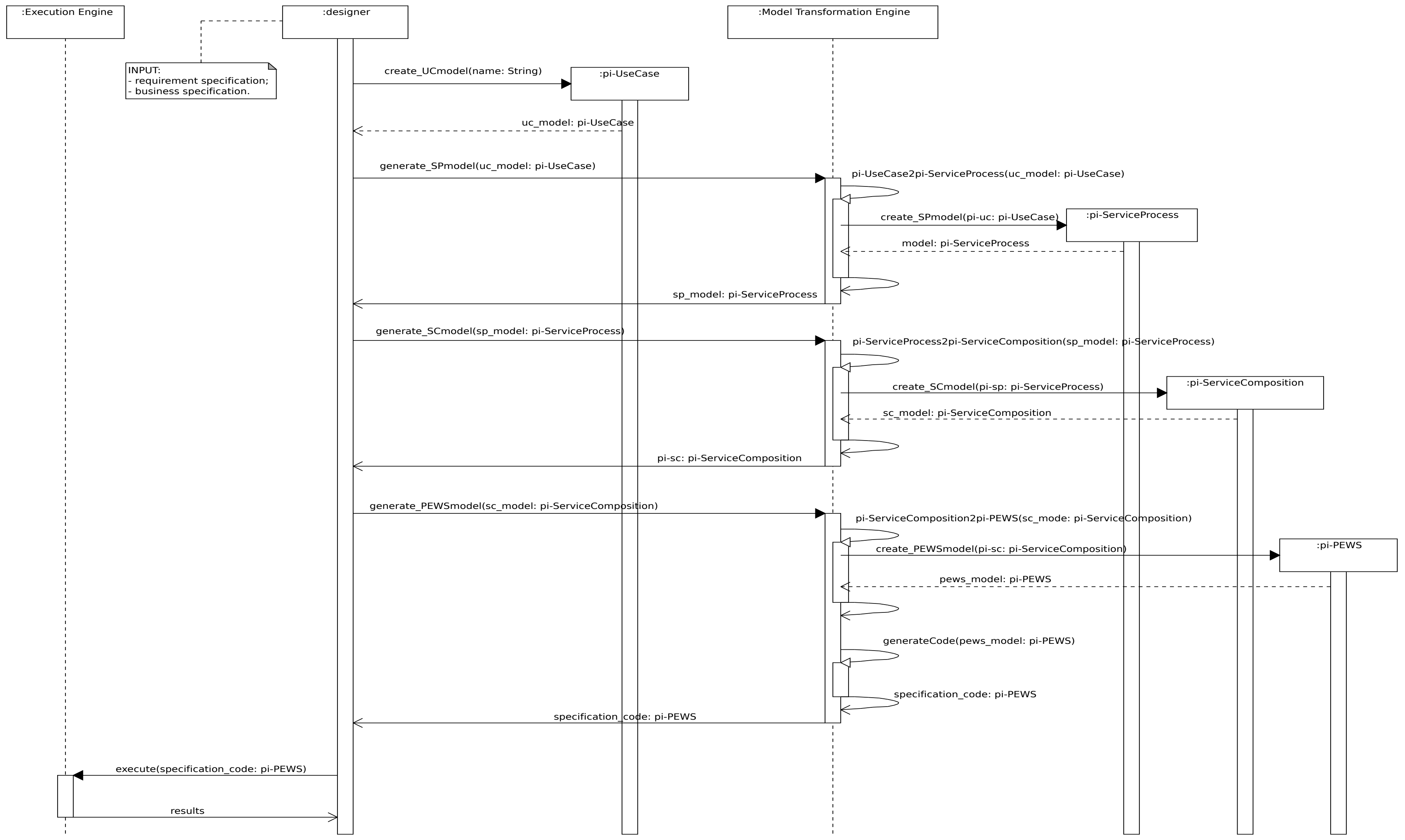

:Execution Engine
:designer
:Model Transformation Engine
INPUT:
- requirement specification;
- business specification.
create_UCmodel(name: String)
:pi-UseCase
uc_model: pi-UseCase
generate_SPmodel(uc_model: pi-UseCase)
pi-UseCase2pi-ServiceProcess(uc_model: pi-UseCase)
create_SPmodel(pi-uc: pi-UseCase)
:pi-ServiceProcess
model: pi-ServiceProcess
sp_model: pi-ServiceProcess
generate_SCmodel(sp_model: pi-ServiceProcess)
pi-ServiceProcess2pi-ServiceComposition(sp_model: pi-ServiceProcess)
create_SCmodel(pi-sp: pi-ServiceProcess)
:pi-ServiceComposition
sc_model: pi-ServiceComposition
pi-sc: pi-ServiceComposition
generate_PEWSmodel(sc_model: pi-ServiceComposition)
pi-ServiceComposition2pi-PEWS(sc_mode: pi-ServiceComposition)
create_PEWSmodel(pi-sc: pi-ServiceComposition)
:pi-PEWS
pews_model: pi-PEWS
generateCode(pews_model: pi-PEWS)
specification_code: pi-PEWS
specification_code: pi-PEWS
execute(specification_code: pi-PEWS)
results

Figure 47: Model Transformation Process.

The generated code can be executed in both, $\pi$-*PEWS* and A-Policy engines[6] These 2 engines are not native components in the $\pi$SOD-M plugin. The environment supports the process design to generate code in any language. New language editor components, like BPEL or XAML, can be easily coupled to the environment. Therewith, it is necessary to add the language meta-model and make the transformation process. Thus, from a $\pi$-*ServiceComposition* model, different models and codes can be generated (not only $\pi$-*PEWS*). This requires only the definition of equivalent meta-models, and the corresponding code transformation rules.

The composition engine manages the life cycle of the composition. Once a composition instance is activated, the engine schedules the composition activities according to the composition control flow. Each activity is seen as a process where the service method call is executed. The execution of an activity has four states: prepared, started, terminated, and failure. The execution of the control flow (sequence, and/or split and join) can also be prepared, started, terminated and raise a failure.

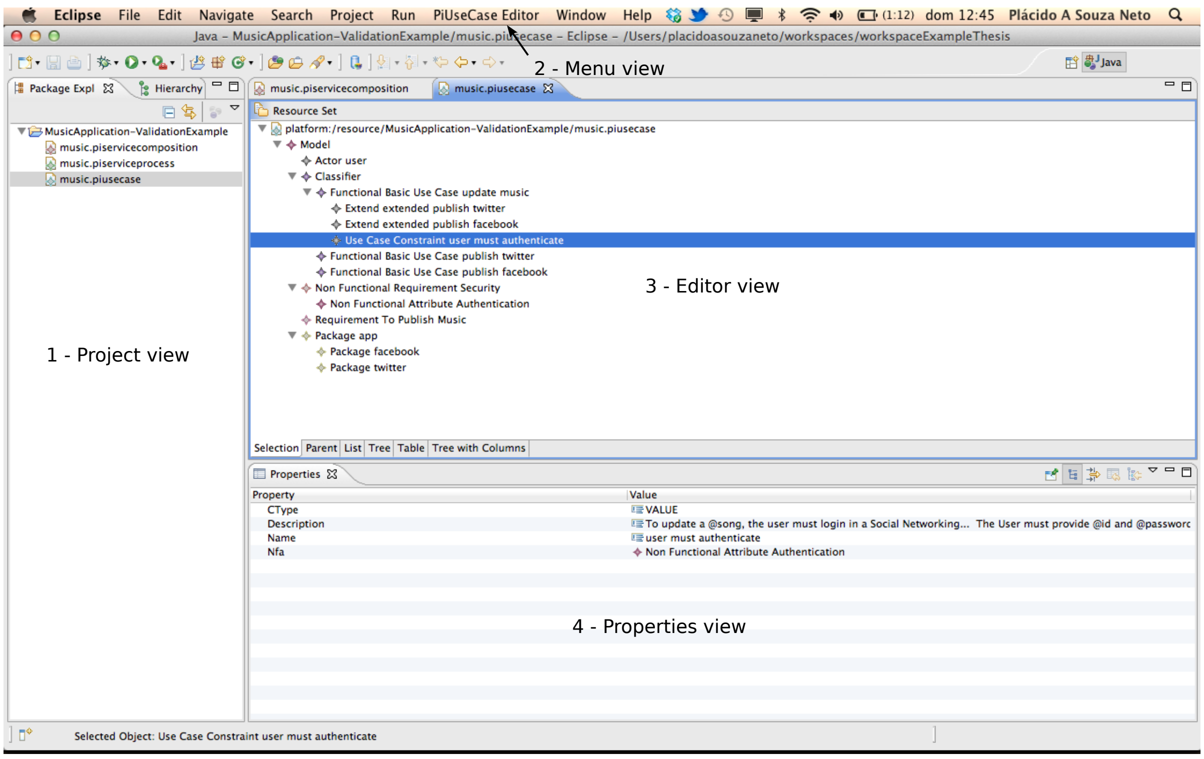
At execution time, the evaluation of policies done by the POLICY manager must be synchronized with the execution of the services composition (i.e., the execution of an activity or a control flow). Policies associated to a scope are activated when the execution of its scope starts. A POLICY will have to be executed only if one or several of its rules is triggered. If several rules are triggered, the *Policy* manager first builds an execution plan that specifies the order in which such rules will be executed according to the strategies defined in the following section. If rules belonging to several policies are triggered then policies are also ordered according to an execution plan. The execution of policies is out of the scope of this thesis, the interested reader can refer to [55] for further details.

## 4.2 Defining Reliable Service Based Applications

The $\pi$SOD-M environment development starts with the creation of a project and then the definition of a $\pi$-*UseCase* model[7], supposing that the business and requirement specification document have been previously completed. Figure 48 presents the views provided by the environment: *Project view, Menu view, Editor view* and *Properties view*.

[6] Both engines are currently under development. The $\pi$-*PEWS* engine is being developed at UFRN anf the A-Policy engine in being developed at Grenoble/France.

[7]To create a model, the user must execute the sequence: *File* > *New* > *Other* > *EMF Model Wizard*, and choose one of the methodology's model.


Eclipse File Edit Navigate Search Project Run PiUseCase Editor Window Help
Java – MusicApplication–ValidationExample/music.piusecase – Eclipse – /Users/placidoasouzaneto/workspaces/workspaceExampleThesis
2 - Menu view
Package Expl
Hierarchy
music.piservicecomposition
music.piusecase
1 - Project view
3 - Editor view
Selection
Parent
List
Tree
Table
Tree with Columns
Properties
4 - Properties view
Java

- MusicApplication–ValidationExample
  - music.piservicecomposition
  - music.piserviceprocess
  - music.piusecase

```
Resource Set
▼ platform:/resource/MusicApplication–ValidationExample/music.piusecase
  ▼ Model
      Actor user
    ▼ Classifier
      ▼ Functional Basic Use Case update music
          Extend extended publish twitter
          Extend extended publish facebook
          Use Case Constraint user must authenticate
        Functional Basic Use Case publish twitter
        Functional Basic Use Case publish facebook
    ▼ Non Functional Requirement Security
        Non Functional Attribute Authentication
      Requirement To Publish Music
    ▼ Package app
        Package facebook
        Package twitter
```

| Property | Value |
|---|---|
| CType | VALUE |
| Description | To update a @song, the user must login in a Social Networking... The User must provide @id and @passworc |
| Name | user must authenticate |
| Nfa | Non Functional Attribute Authentication |

Selected Object: Use Case Constraint user must authenticate

Figure 48: $\pi$SOD-M Eclipse Plugin Environment.

### 4.2.1 $\pi$-*UseCase* Model

The goal of creating a $\pi$-*UseCase* is to represent the functions and system services described in the requirements specification and business specification documents, which are the requirements input for this phase. In accordance to the process described in figure 47, the designer receives the documents as input and creates the $\pi$-*UseCase* model. With this model created, the transformation component generates the $\pi$-*ServiceProcess* model as output.

To create the $\pi$-*UseCase* model, it is necessary choose the root element (*Model*) as the starting point of modeling, during the process of create a "*new .piusecase file*" ( using the sequence, *File* > *New* > *Other* > *EMF Model Wizard*). From this point the model is created as a tree of elements with its specific references, as shown in Figure 49, which shows the model elements and its equivalence in the graphical $\pi$-UseCase model, each element is built in an iterative way and must obey the hierarchy and its relationships.

Each model element has a number of childs or siblings to which it is related to, *e.g.*, an USE CASE relates CONSTRAINT, EXTEND and INCLUDE elements, as well as NFRS relates NFAS. Figure 49 shows how to create the model elements for the "*to publish music*" scenario.

*Example:* Items 1 and 2 in Figure 49 show how to create an ACTOR and a REQUIREMENT, item 3 presents the *update music* USE CASE and how to create a CONSTRAINT related to this

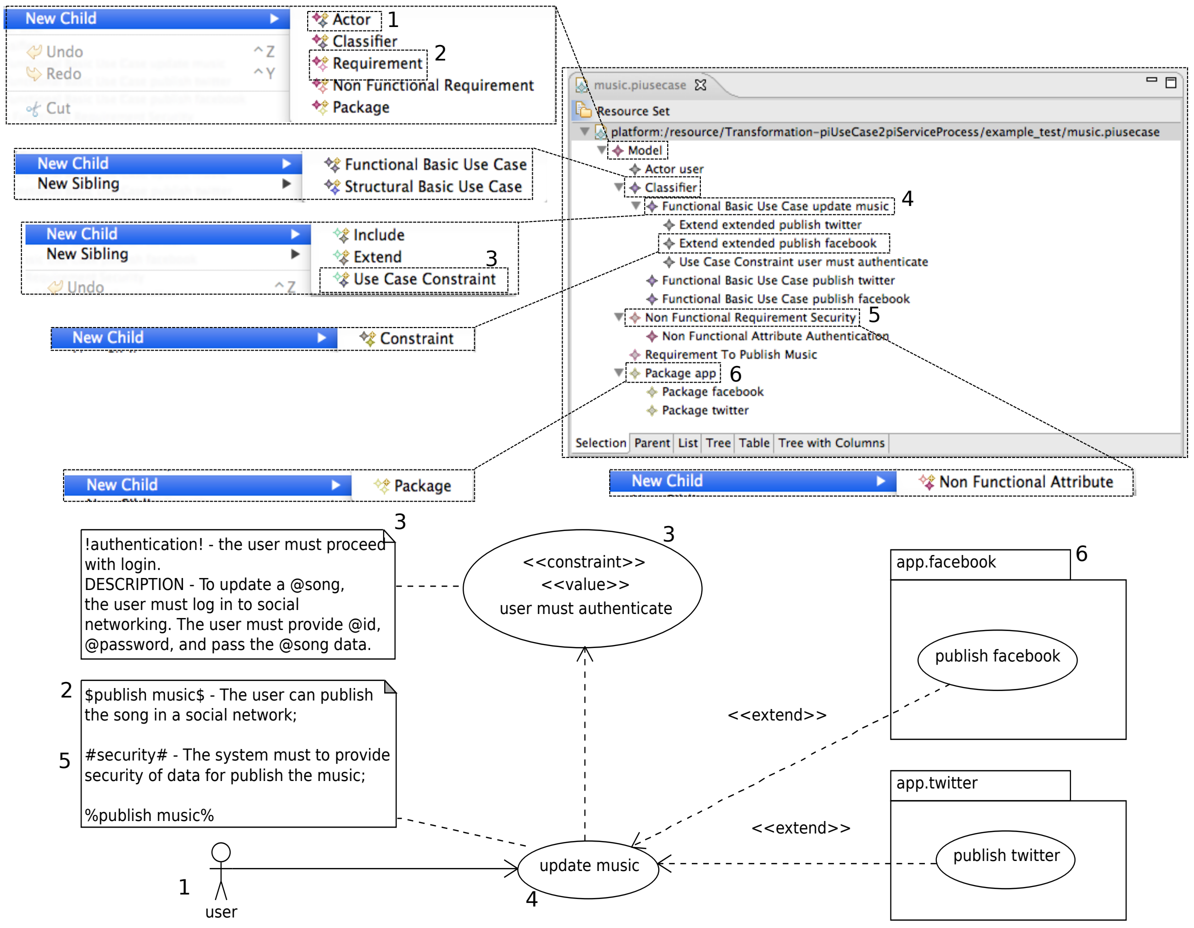

New Child
Undo ^Z
Redo ^Y
Cut
Actor 1
Classifier
Requirement 2
Non Functional Requirement
Package
New Child
New Sibling
Functional Basic Use Case
Structural Basic Use Case
New Child
New Sibling
Undo ^Z
Include
Extend 3
Use Case Constraint
New Child
Constraint
music.piusecase
Resource Set
platform:/resource/Transformation-piUseCase2piServiceProcess/example_test/music.piusecase
Model
Actor user
Classifier
Functional Basic Use Case update music 4
Extend extended publish twitter
Extend extended publish facebook
Use Case Constraint user must authenticate
Functional Basic Use Case publish twitter
Functional Basic Use Case publish facebook
Non Functional Requirement Security 5
Non Functional Attribute Authentication
Requirement To Publish Music
Package app 6
Package facebook
Package twitter
Selection Parent List Tree Table Tree with Columns
New Child
Package
New Child
Non Functional Attribute
3
!authentication! - the user must proceed with login.
DESCRIPTION - To update a @song, the user must log in to social networking. The user must provide @id, @password, and pass the @song data.
3
<<constraint>>
<<value>>
user must authenticate
app.facebook 6
publish facebook
2
$publish music$ - The user can publish the song in a social network;
5
#security# - The system must to provide security of data for publish the music;
%publish music%
<<extend>>
app.twitter
<<extend>>
publish twitter
1
user
update music
4

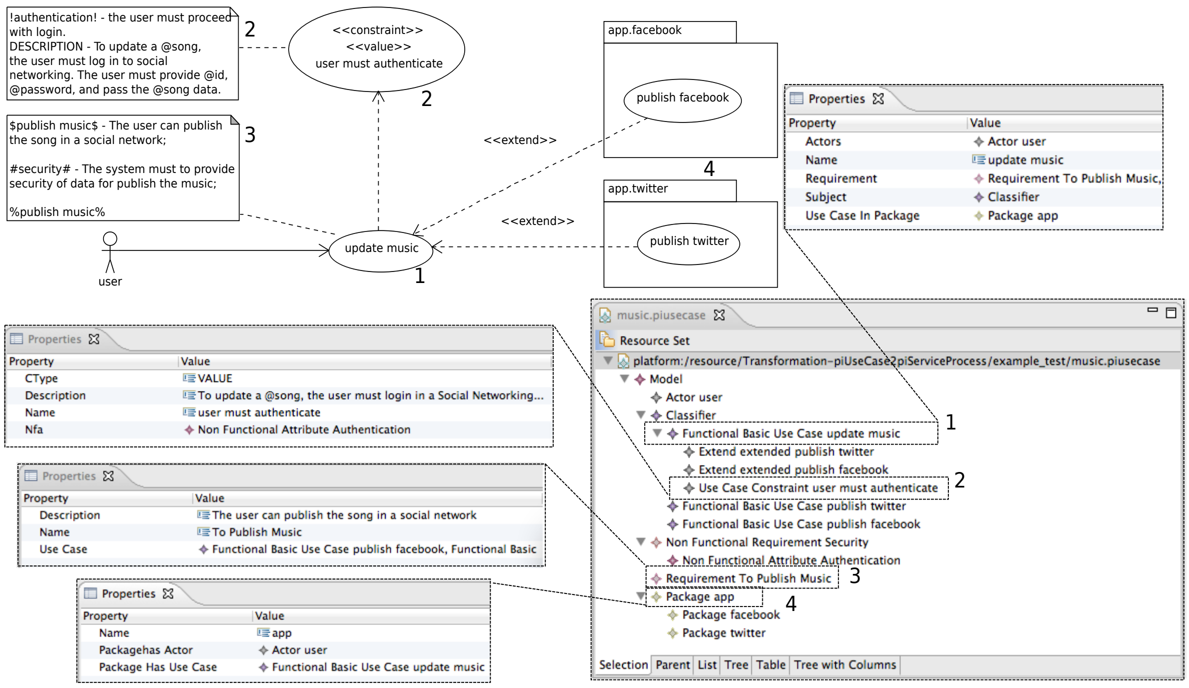
Figure 49: $\pi$-UseCase Model Definition in $\pi$SOD-M Eclipse Plugin.

element. Item 4 makes reference to a CONSTRAINT and finally, the items 5 and 6 are equivalent to a NFR and a PACKAGE elements, respectively. Each item in the model refers to a equivalent element in the graphical model. □

Figure 50 presents some configuration properties. After creating an element, it is necessary to set its properties. All elements have properties that describes their relationship and specification values. These properties are used to give specification details, as well as future transformations.

*Example:* Item 1 in Figure 50 shows the USE CASE element properties for the "*to publish music*" application. After the creation of a use case it is necessary to create the actor, its *name*, which *requirement* the *use case* belongs, and the respective *package*. Item 2 shows the properties of a CONSTRAINT element. In a constraint, its type must be explicit (*VALUE, BUSINESS* or *EXCEPTIONALBEHAVIOUR*), its *description*, with all candidate *variables* described with an '@', its *name* and which NON-FUNCTIONAL ATTRIBUTES element this constraint is related to, *e.g.*, *Authentication*, is an attribute of the NON-FUNCTIONAL REQUIREMENTS, *Security*. Items

3 and 4 show the "*To Publish Music*" requirement properties and package "*app*", respectively. The properties of each element obey the elements described in the $\pi$-*UseCase* meta-model.


!authentication! - the user must proceed with login.
DESCRIPTION - To update a @song, the user must log in to social networking. The user must provide @id, @password, and pass the @song data.
2
<<constraint>>
<<value>>
user must authenticate
2
$publish music$ - The user can publish the song in a social network;
#security# - The system must to provide security of data for publish the music;
%publish music%
3
app.facebook
publish facebook
<<extend>>
4
app.twitter
publish twitter
<<extend>>
user
update music
1

Properties

| Property | Value |
|---|---|
| Actors | Actor user |
| Name | update music |
| Requirement | Requirement To Publish Music, |
| Subject | Classifier |
| Use Case In Package | Package app |

Properties

| Property | Value |
|---|---|
| CType | VALUE |
| Description | To update a @song, the user must login in a Social Networking... |
| Name | user must authenticate |
| Nfa | Non Functional Attribute Authentication |

Properties

| Property | Value |
|---|---|
| Description | The user can publish the song in a social network |
| Name | To Publish Music |
| Use Case | Functional Basic Use Case publish facebook, Functional Basic |

Properties

| Property | Value |
|---|---|
| Name | app |
| Packagehas Actor | Actor user |
| Package Has Use Case | Functional Basic Use Case update music |

music.piusecase

```
Resource Set
▼ platform:/resource/Transformation-piUseCase2piServiceProcess/example_test/music.piusecase
  ▼ Model
      Actor user
    ▼ Classifier
      ▼ Functional Basic Use Case update music                1
          Extend extended publish twitter
          Extend extended publish facebook
          Use Case Constraint user must authenticate          2
        Functional Basic Use Case publish twitter
        Functional Basic Use Case publish facebook
    ▼ Non Functional Requirement Security
        Non Functional Attribute Authentication
      Requirement To Publish Music       3
    ▼ Package app          4
        Package facebook
        Package twitter
Selection | Parent | List | Tree | Table | Tree with Columns
```

Figure 50: $\pi$-*UseCase* Properties in $\pi$SOD-M Eclipse Plugin.

### 4.2.2 $\pi$-*ServiceProcess* Models

The goal of creating a $\pi$-*ServiceProcess* model is to represent the system execution flow. The designer receives the $\pi$-*UseCase* model as input to generate the $\pi$-*ServiceProcess* model. After the $\pi$-*ServiceProcess* model be generated, the designer calls again the model transformation component to generate the $\pi$-*ServiceComposition* model as output (figure 47).

To create the $\pi$-*ServiceProcess* model, it is necessary choose the root object (*Service Process*) as the starting point of modeling, during the process of create a "*new .piserviceprocess file*" option. From this point on, the model is created as a tree with its specific references.

As this model is a refinement of the concepts described in the previous model, its information is part of the information and properties of the $\pi$-*UseCase* model, but they focus on the process workflow and application functions.

*Example:* Figure 51 shows the complete model of the example scenario and its main components. From the root element the user can create the following elements: SERVICE ACTIVITY, OBJECT FLOW, CONTROL FLOW, CONTRACT and NON-FUNCTIONAL ATTRIBUTE. In Figure 51, item marked with 1 highlights the creation of the CONTROL FLOW . Items 2 and 3 show the elements FORK NODE and JOIN NODE, respectively, essential for the execution flow description. Items 4 show the ACTION element, which describes the *download music, listen music* and *publish twitter* actions and finally, items 5 highlight the ASSERTION element, which is used to describe the restrictions over each ACTION. □

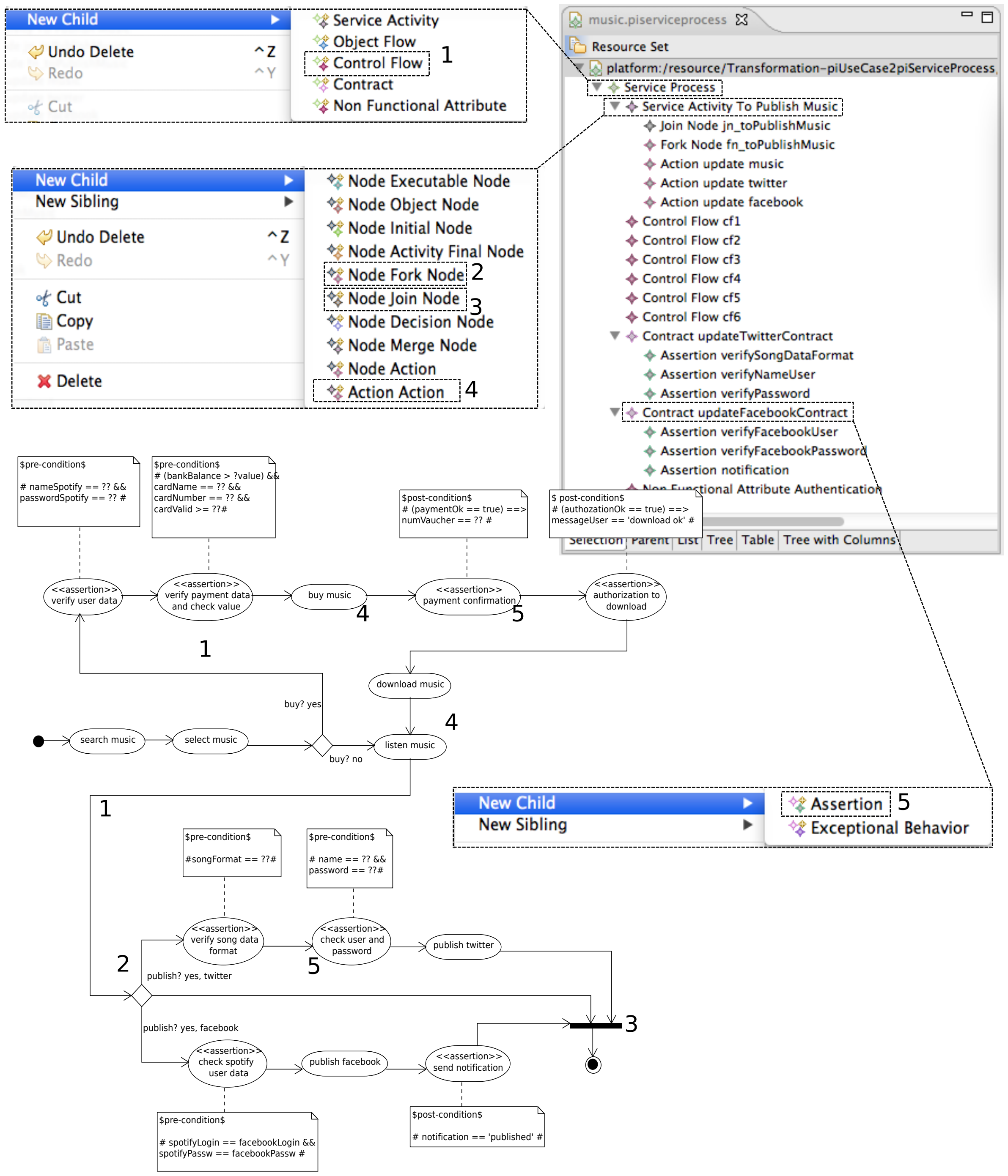
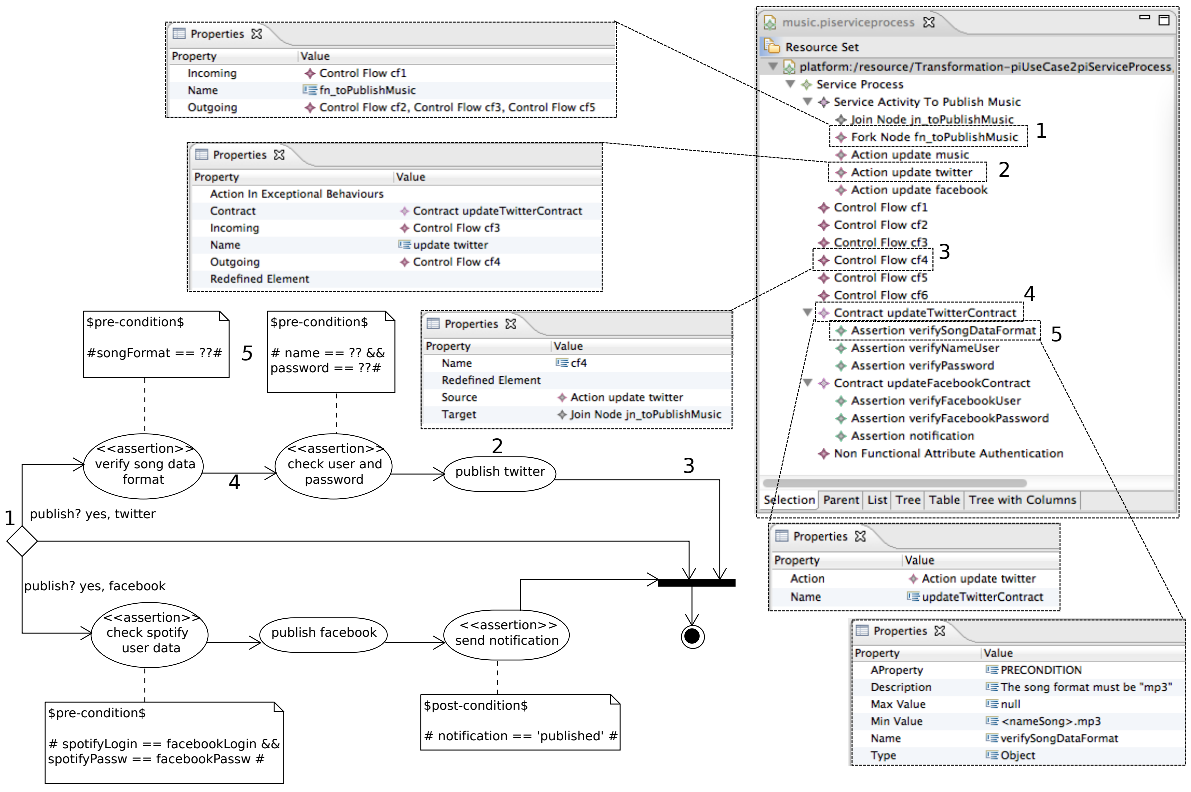

New Child
Undo Delete ^Z
Redo ^Y
Cut
Service Activity
Object Flow
Control Flow
1
Contract
Non Functional Attribute
music.piserviceprocess
Resource Set
platform:/resource/Transformation-piUseCase2piServiceProcess
Service Process
Service Activity To Publish Music
Join Node jn_toPublishMusic
Fork Node fn_toPublishMusic
Action update music
Action update twitter
Action update facebook
Control Flow cf1
Control Flow cf2
Control Flow cf3
Control Flow cf4
Control Flow cf5
Control Flow cf6
Contract updateTwitterContract
Assertion verifySongDataFormat
Assertion verifyNameUser
Assertion verifyPassword
Contract updateFacebookContract
Assertion verifyFacebookUser
Assertion verifyFacebookPassword
Assertion notification
Non Functional Attribute Authentication
Selection Parent List Tree Table Tree with Columns
New Child
New Sibling
Undo Delete ^Z
Redo ^Y
Cut
Copy
Paste
Delete
Node Executable Node
Node Object Node
Node Initial Node
Node Activity Final Node
Node Fork Node
2
Node Join Node
3
Node Decision Node
Node Merge Node
Node Action
Action Action
4
$pre-condition$
# nameSpotify == ?? && passwordSpotify == ?? #
$pre-condition$
# (bankBalance > ?value) && cardName == ?? && cardNumber == ?? && cardValid >= ??#
$post-condition$
# (paymentOk == true) ==> numVaucher == ?? #
$ post-condition$
# (authozationOk == true) ==> messageUser == 'download ok' #
<<assertion>> verify user data
<<assertion>> verify payment data and check value
buy music
4
<<assertion>> payment confirmation
5
<<assertion>> authorization to download
1
download music
buy? yes
search music
select music
buy? no
listen music
4
1
New Child
New Sibling
Assertion
5
Exceptional Behavior
$pre-condition$
#songFormat == ??#
$pre-condition$
# name == ?? && password == ??#
<<assertion>> verify song data format
<<assertion>> check user and password
publish twitter
2
5
publish? yes, twitter
publish? yes, facebook
3
<<assertion>> check spotify user data
publish facebook
<<assertion>> send notification
$pre-condition$
# spotifyLogin == facebookLogin && spotifyPassw == facebookPassw #
$post-condition$
# notification == 'published' #

Figure 51: $\pi$-*ServiceProcess* Model Definition in $\pi$SOD-M Eclipse Plugin.

The designer must also configure the properties of each $\pi$-*ServiceProcess* model element to complete it by specifying all constraints that compose the contracts and the application process.

Figure 52 presents the properties of the main model elements for our example scenario. These properties complement the modeling of the execution flow.


$pre-condition$
#songFormat == ??#
5
$pre-condition$
# name == ?? && password == ??#
<<assertion>> verify song data format
4
<<assertion>> check user and password
2
publish twitter
3
1
publish? yes, twitter
publish? yes, facebook
<<assertion>> check spotify user data
publish facebook
<<assertion>> send notification
$pre-condition$
# spotifyLogin == facebookLogin && spotifyPassw == facebookPassw #
$post-condition$
# notification == 'published' #
music.piserviceprocess
Resource Set
platform:/resource/Transformation-piUseCase2piServiceProcess
Service Process
Service Activity To Publish Music
Join Node jn_toPublishMusic
Fork Node fn_toPublishMusic
1
Action update music
Action update twitter
2
Action update facebook
Control Flow cf1
Control Flow cf2
Control Flow cf3
3
Control Flow cf4
Control Flow cf5
Control Flow cf6
4
Contract updateTwitterContract
Assertion verifySongDataFormat
5
Assertion verifyNameUser
Assertion verifyPassword
Contract updateFacebookContract
Assertion verifyFacebookUser
Assertion verifyFacebookPassword
Assertion notification
Non Functional Attribute Authentication
Selection
Parent
List
Tree
Table
Tree with Columns
Properties

| Property | Value |
|---|---|
| Incoming | Control Flow cf1 |
| Name | fn_toPublishMusic |
| Outgoing | Control Flow cf2, Control Flow cf3, Control Flow cf5 |

| Property | Value |
|---|---|
| Action In Exceptional Behaviours | |
| Contract | Contract updateTwitterContract |
| Incoming | Control Flow cf3 |
| Name | update twitter |
| Outgoing | Control Flow cf4 |
| Redefined Element | |

| Property | Value |
|---|---|
| Name | cf4 |
| Redefined Element | |
| Source | Action update twitter |
| Target | Join Node jn_toPublishMusic |

| Property | Value |
|---|---|
| Action | Action update twitter |
| Name | updateTwitterContract |

| Property | Value |
|---|---|
| AProperty | PRECONDITION |
| Description | The song format must be "mp3" |
| Max Value | null |
| Min Value | <nameSong>.mp3 |
| Name | verifySongDataFormat |
| Type | Object |

Figure 52: $\pi$-*ServiceProcess* Properties in $\pi$SOD-M Eclipse Plugin.

*Example:* Items 1 in Figure 52 describe the properties of a FORK NODE, which is associated with EDGE ACTIVITIES (CONTROL FLOW and OBJECT FLOW) that connect the ACTION elements. For example, the FORK NODE *fn_toPublishMusic* has an input edge (*cf1*) and three output streams edges (*cf2, cf3* and *cf5*). All these elements are CONTROL FLOW entities (items 3). Items 2 show the ACTION properties, and items 4 and 5 describe a CONTRACT and its ASSERTIONS properties, respectively. A CONTRACT has a *name* and the information of an ACTION. ASSERTIONS are CONTRACT's child element. Each CONTRACT can have many ASSERTIONS. Each ASSERTION has six properties, they are: (i) *AProperty*, which describes the runtime verification execution time, (ii) *description*, (iii) *variable name*, (iv and v) a maximum and minimum allowed to be checked (*MaxValue* and *MinValue*) and the variable (vi) *type*. In cases of non numeric types, *e.g. Short Int, Flot* and *Double*, only *MinValue* property value is considered property and*MaxValue* is described as *null*. □

### 4.2.3 *π-ServiceComposition* Models

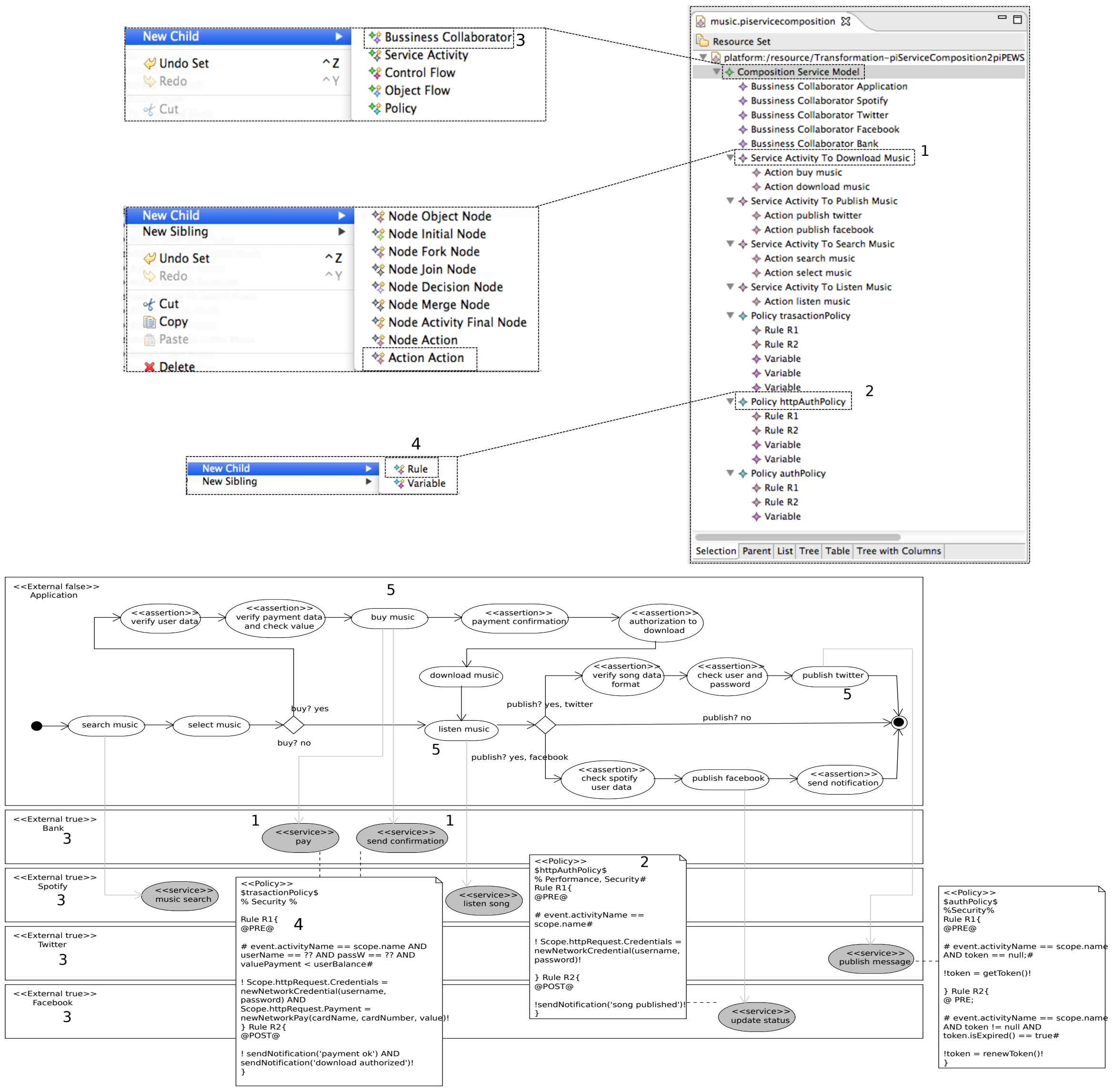
The goal of creating a *π-ServiceComposition* model is to represent the system service compositions and to reference the external services that are called by the application. The designer receives the *π-ServiceProcess* model as input to generate the *π-ServiceComposition* model. After the *π-ServiceComposition* model is generated, the designer calls again the model transformation component to generate the *π-PEWS* model as output (figure 47). The output model describes the system specification in a specific platform.

To create the *π-ServiceComposition* model, it is necessary to choose the root object (*Composition Service Model*) as the starting point of modeling, during the process of creating a "new .piservicecomposition file" option. From this point on, the model is created as a tree with its specific references.

Recall that the *π-ServiceComposition* model is a refinement of *π-ServiceProcess* model, most elements are the same, except for BUSINESS COLLABORATOR, POLICY, RULES and VARIABLE. Thus, we will detail these elements in the model editor description.

*Example:* Figure 53 shows the relationship of these elements for our example scenario. Each ACTION has an specific BUSINESS COLLABORATOR (item 3), this elements express the external service provider, such as Facebook, Twitter or Bank. Another example of a BUSINESS COLLABORATOR definition is a WSDL specification. The WSDL namespaces represents a kind of BUSINESS COLLABORATOR. Besides BUSINESS COLLABORATOR element, the main children root element (*Composition Service Model*) are: POLICY and SERVICE ACTIVITY (items 1), and each POLICY (items 2) is directly related with SERVICE ACTIVITIES. From a POLICY it is possible to create a set of RULES (item 4). Figure 53 also presents an equivalent graphic model of the application. In this figure, the *Application*'s BUSINESS COLLABORATOR presents only the main ACTIONS (*buy music, listen music, publish Facebook* and *publish twitter*) that are related with a SERVICE ACTIVITIES that have one POLICY. The *Application*'s BUSINESS COLLABORATOR model is the same presented in the previous section (*π-ServiceProcess* model).

The properties of *π-ServiceComposition* elements are configured as the same way the other two previous editors, and the properties described in the *π-ServiceComposition* meta-model (Figure 29). Thus, to configure the properties of this model, it is necessary simply choose the desired element and to modify its values.


New Child
Undo Set ^Z
Redo ^Y
Cut
Bussiness Collaborator
Service Activity
Control Flow
Object Flow
Policy
3
New Child
New Sibling
Undo Set ^Z
Redo ^Y
Cut
Copy
Paste
Delete
Node Object Node
Node Initial Node
Node Fork Node
Node Join Node
Node Decision Node
Node Merge Node
Node Activity Final Node
Node Action
Action Action
4
New Child
New Sibling
Rule
Variable
music.piservicecomposition
Resource Set
platform:/resource/Transformation-piServiceComposition2piPEWS
Composition Service Model
Bussiness Collaborator Application
Bussiness Collaborator Spotify
Bussiness Collaborator Twitter
Bussiness Collaborator Facebook
Bussiness Collaborator Bank
Service Activity To Download Music
1
Action buy music
Action download music
Service Activity To Publish Music
Action publish twitter
Action publish facebook
Service Activity To Search Music
Action search music
Action select music
Service Activity To Listen Music
Action listen music
Policy trasactionPolicy
Rule R1
Rule R2
Variable
Variable
Variable
2
Policy httpAuthPolicy
Rule R1
Rule R2
Variable
Variable
Policy authPolicy
Rule R1
Rule R2
Variable
Selection Parent List Tree Table Tree with Columns
<<External false>> Application
<<assertion>> verify user data
<<assertion>> verify payment data and check value
5
buy music
<<assertion>> payment confirmation
<<assertion>> authorization to download
download music
<<assertion>> verify song data format
<<assertion>> check user and password
publish twitter
5
buy? yes
publish? yes, twitter
search music
select music
listen music
publish? no
buy? no
5
publish? yes, facebook
<<assertion>> check spotify user data
publish facebook
<<assertion>> send notification
<<External true>> Bank
3
1
<<service>> pay
<<service>> send confirmation
1
<<External true>> Spotify
3
<<service>> music search
<<service>> listen song
<<External true>> Twitter
3
<<service>> publish message
<<External true>> Facebook
3
<<service>> update status
4
2

```
<<Policy>>
$trasactionPolicy$
% Security %

Rule R1{
@PRE@

# event.activityName == scope.name AND
userName == ?? AND passW == ?? AND
valuePayment < userBalance#

! Scope.httpRequest.Credentials =
newNetworkCredential(username,
password) AND
Scope.httpRequest.Payment =
newNetworkPay(cardName, cardNumber, value)!
} Rule R2{
@POST@

! sendNotification('payment ok') AND
sendNotification('download authorized')!
}
```

```
<<Policy>>
$httpAuthPolicy$
% Performance, Security#
Rule R1{
@PRE@

# event.activityName ==
scope.name#

! Scope.httpRequest.Credentials =
newNetworkCredential(username,
password)!

} Rule R2{
@POST@

!sendNotification('song published')!
}
```

```
<<Policy>>
$authPolicy$
%Security%
Rule R1{
@PRE@

# event.activityName == scope.name
AND token == null;#

!token = getToken()!

} Rule R2{
@ PRE;

# event.activityName == scope.name
AND token != null AND
token.isExpired() == true#

!token = renewToken()!
}
```

Figure 53: *π-ServiceComposition* Model Definition in *π*SOD-M Eclipse Plugin.

## 4.2.4 *π-PEWS* Models

The goal of creating a *π-PEWS* model is to represent the application in a specific platform. The designer receives the *π-ServiceComposition* model as input to generate the *π-PEWS* model. After the *π-PEWS* model be generated, the designer calls again the model transformation component to generate the *π-PEWS* specification code as output (figure 47). This code will be executed.

To create the *π-PEWS* model, it is necessary to choose the root object (*PEWS Spec*) as the starting point of modeling. From this point on, the model is created as a tree with its specific references.

The *π-PEWS* meta-model is a representation of the *π-PEWS* language. Each *π-PEWS* model is a program/specification written in that language. The elements described in this model represent parts of language's grammar constructs. Each entity in the model represents a piece of $\pi$-*PEWS* code.

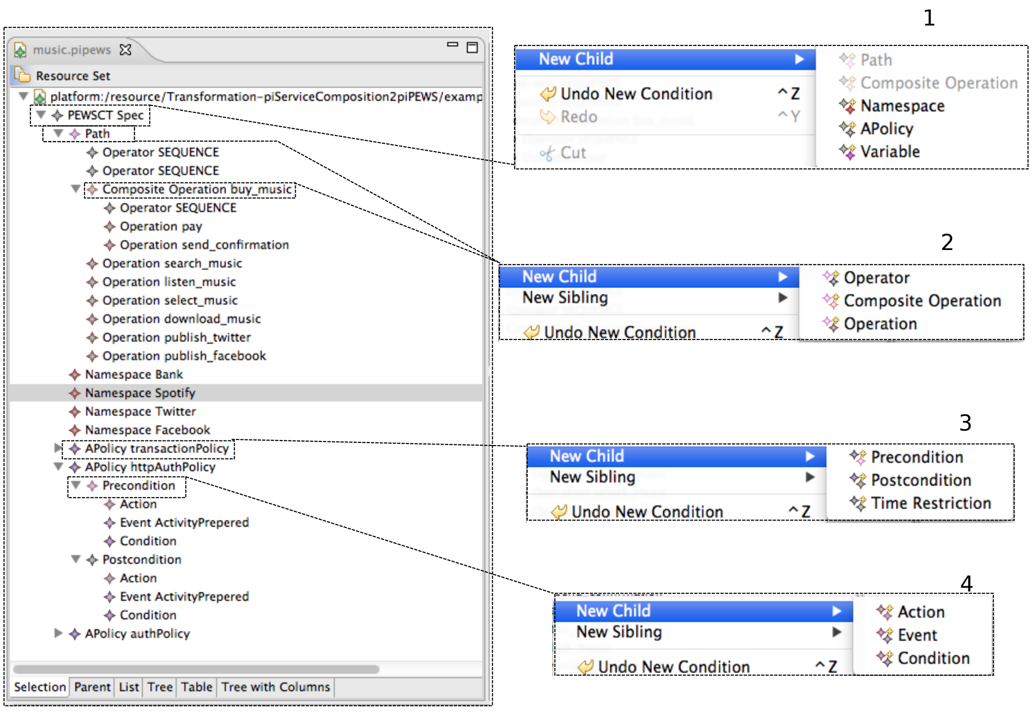


Figure 54: $\pi$-*PEWS* Model Definition in $\pi$SOD-M Eclipse Plugin.

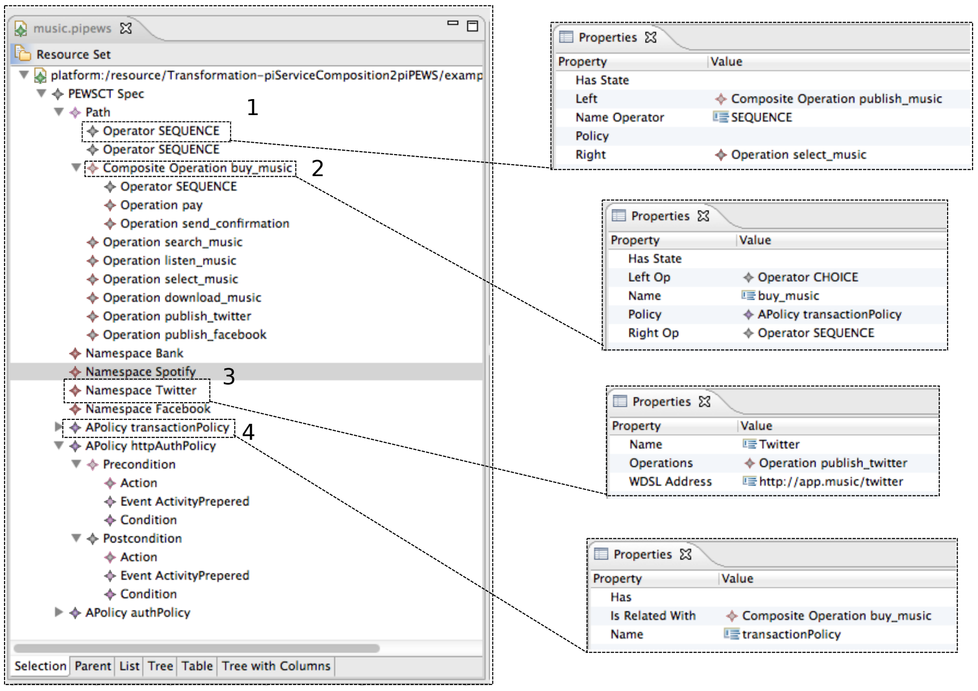
Figures 54 and 55 show how the elements that compose a $\pi$-*PEWS* model can be specified in our tool. This model is generated automatically from the $\pi$-*ServiceComposition* model.

$\pi$-*PEWS* is the last model generated before the code generation. At this stage, it is spected that the designer proceeds with a general (manual) check of consistency of the model. It is important to remark that the environment alone does not replace the modeling work required to design and develop services based applications.

## 4.3 Extending the Environment

Both, the $\pi$SOD-M environment and methodology can be extended, in order to improve the components that describe and implement the methodology. Extensions can be done in two different levels: (i) adding new models to the existing infra-structure, and (ii) considering more abstract levels.

The extension may be in terms of language: new meta-models for other languages can be described and coupled to the environment. The extension process should take place as follows:



```
platform:/resource/Transformation-piServiceComposition2piPEWS/examp
  PEWSCT Spec
    Path
      Operator SEQUENCE
      Operator SEQUENCE
      Composite Operation buy_music
        Operator SEQUENCE
        Operation pay
        Operation send_confirmation
      Operation search_music
      Operation listen_music
      Operation select_music
      Operation download_music
      Operation publish_twitter
      Operation publish_facebook
    Namespace Bank
    Namespace Spotify
    Namespace Twitter
    Namespace Facebook
    APolicy transactionPolicy
    APolicy httpAuthPolicy
      Precondition
        Action
        Event ActivityPrepered
        Condition
      Postcondition
        Action
        Event ActivityPrepered
        Condition
    APolicy authPolicy
```

Properties

| Property | Value |
|---|---|
| Has State | |
| Left | Composite Operation publish_music |
| Name Operator | SEQUENCE |
| Policy | |
| Right | Operation select_music |

Properties

| Property | Value |
|---|---|
| Has State | |
| Left Op | Operator CHOICE |
| Name | buy_music |
| Policy | APolicy transactionPolicy |
| Right Op | Operator SEQUENCE |

Properties

| Property | Value |
|---|---|
| Name | Twitter |
| Operations | Operation publish_twitter |
| WDSL Address | http://app.music/twitter |

Properties

| Property | Value |
|---|---|
| Has | |
| Is Related With | Composite Operation buy_music |
| Name | transactionPolicy |

Figure 55: $\pi$-*PEWS* Model Properties in $\pi$SOD-M Eclipse Plugin.

new languages meta-models may be designed and coupled in the environment architecture (such as BPEL, XAML, WSDL and XLANG). After creating the desired meta-model, a mapping must be done. The mapping must respect the $\pi$-*ServiceComposition* meta-model. It is also necessary to describe the rules for code generation, using a code generator engine such Acceleo.

Considering the more abstract level of the methodology, new meta-models can be described. For instance, the computing independent level (CIM level) of SOD-M may be added to our methodology to describe the requirements and business restrictions in terms of models.

## 4.4 Conclusion

This chapter introduced the implementation of the $\pi$SOD-M methodology environment. We also presented a representation for our model description. The implementation includes all (i) meta-models of $\pi$SOD-M, (ii) editors for each model for the applications being developed, and (iii) the plugins for the transformations defined by the methodology. An example was also presented to describe the environment features.

# 5 Validation

"*Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.*"

Donald E. Knuth

This chapter describes the experimentation we conducted for validating the $\pi$SOD-M methodology. The chapter presents three case studies. The first two case studies concern simplified service-based applications that focus on the application's logic and explicit business rules. They show how $\pi$SOD-M can be used for developing applications and exemplify the consideration of non-functional aspects along the whole development process. The last case study (*GesIMED* [36] application) aims to perform a qualitative evaluation of our approach, analyzing the main differences and particularities between SOD-M and $\pi$SOD-M.

The *IEEE Standard Computer Dictionary* [53] defines validation as *the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements*. If we apply this definition to a methodology: *Validation is the process of evaluating a method during or at the end of the development process to determine whether it satisfies specified requirements and eases the development of a system.* This chapter validates the methodology proposed in this thesis in order to verify whether it promotes a better development of reliable services composition applications.

The validation process has been performed in a continuous and progressive approach along the definition of $\pi$SOD-M.

This chapter is organized as follows: section 5.1 shows *To Listen Music* case study, section 5.2 introduces the case study *Crime Map* and section 5.3 introduces the case study *GesIMED* that we used for validating the methodology. Section 5.4 presents the lessons learned with the validation process. Finally, Section 5.5 concludes the chapter and discusses on validation results.

## 5.1 Case Study 1: To Publish Music

An organization wants to provide the service-based application that monitors the music a person is listening during some periods of time and sends the song title to this person's Twitter or Facebook accounts. Thus, this social network user will have her status synchronized in Twitter or Facebook. The scenario begins by get through the music service Spotify for fetching the user's musical status. The social network services are then contacted to modify the user's status with the music information. The user can also download a specific music, he can proceed with the download process, which includes the payment process.

The user can download a music, after having proceed with the payment via PayPal or credit card. All services are available via Spotify and this application needs to interact with Spotify users, so they can listen music, publish them on Facebook and Twitter, or buy music online. The scenario to monitor and publish songs has the User and Spotify actors.

The functionality of the application are: *Search music; Choose music; Download music; Listen Music; Buy music* and *Publish music*.

### 5.1.1 $\pi$-UseCase Model

Figure 56 shows the main part of the use case model for the *To Publish Music* application[1]. The $\pi$-UseCase model describes the system functions and constraints. Figures 57 and 58 detail the model and configuration properties for the described example (actual view taken from the $\pi$SOD-M environment). Three restrictions are specified on the use cases described by the model, two on the *buy music* use case and a restriction on the *pay* use case. These restrictions are value and business constraints. *Buy music* restrictions are related to secure connection and to ensure that the user needs to have an account in the music network. They are marked using . . . , figures 56, 57 and 58. Notice that those NFRs and constraints will be the base to define assertions and contracts by the transformations of models.

Figures 57 and 58 present the update music model and show how the properties are defined in the $\pi$SOD-M environment. This example describes the properties over the *update music*

[1]The complete $\pi$-*UseCase* model can be found in appendix D.

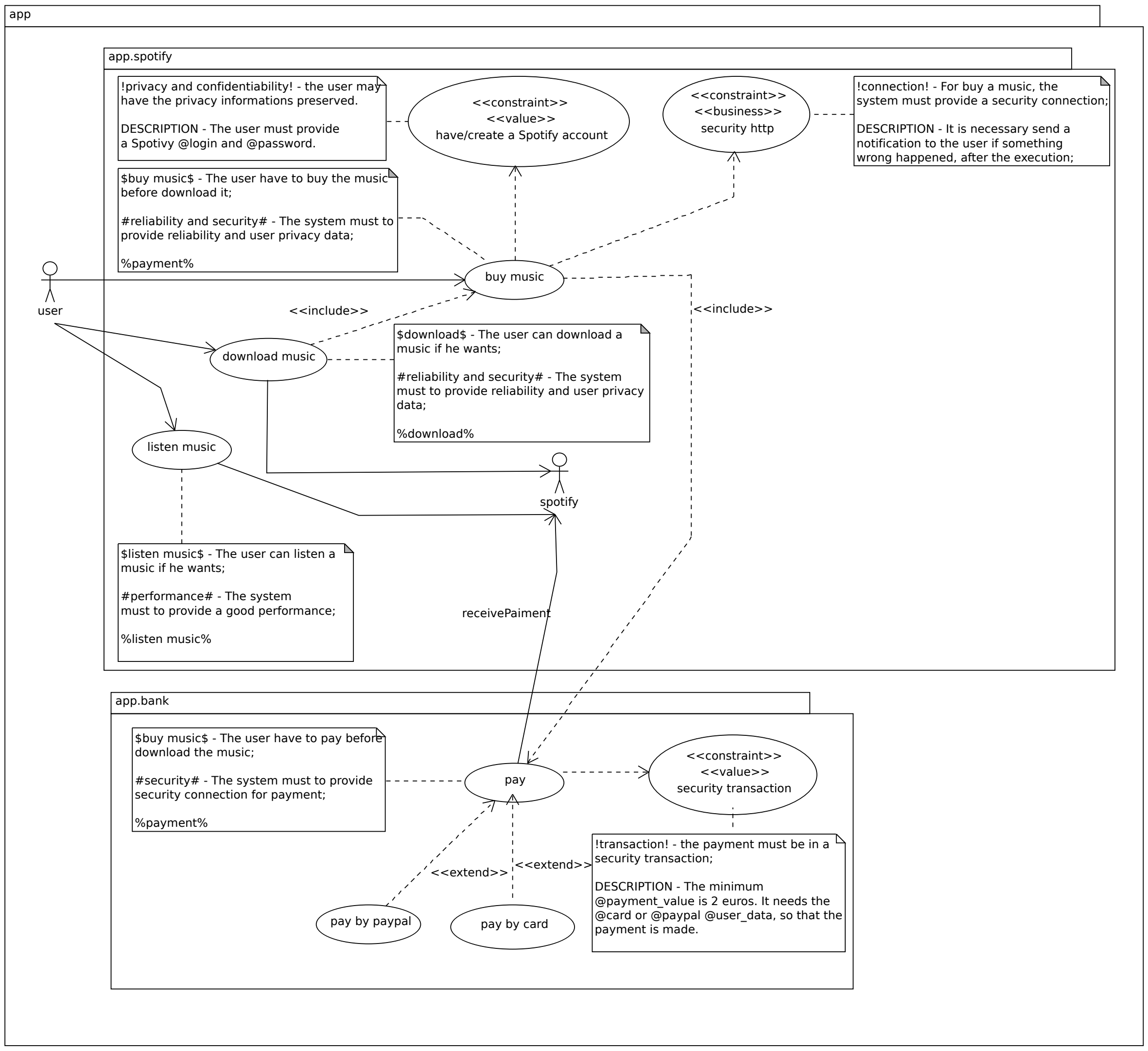

app
app.spotify
!privacy and confidentiability! - the user may have the privacy informations preserved.
DESCRIPTION - The user must provide a Spotivy @login and @password.
<<constraint>> <<value>> have/create a Spotify account
<<constraint>> <<business>> security http
!connection! - For buy a music, the system must provide a security connection;
DESCRIPTION - It is necessary send a notification to the user if something wrong happened, after the execution;
$buy music$ - The user have to buy the music before download it;
#reliability and security# - The system must to provide reliability and user privacy data;
%payment%
user
buy music
<<include>>
<<include>>
download music
$download$ - The user can download a music if he wants;
#reliability and security# - The system must to provide reliability and user privacy data;
%download%
listen music
spotify
$listen music$ - The user can listen a music if he wants;
#performance# - The system must to provide a good performance;
%listen music%
receivePaiment
app.bank
$buy music$ - The user have to pay before download the music;
#security# - The system must to provide security connection for payment;
%payment%
pay
<<constraint>> <<value>> security transaction
!transaction! - the payment must be in a security transaction;
DESCRIPTION - The minimum @payment_value is 2 euros. It needs the @card or @paypal @user_data, so that the payment is made.
<<extend>>
<<extend>>
pay by paypal
pay by card

Figure 56: *To Publish Music* π-UseCase.

(item 4, figure 57) use case and the restriction over the authentication process (item 3, figure 57). To update a music in a social network, the user must be authenticated.

The modeling of the application properties using the environment πSOD-M are the same presented in the π-UseCase model, as described in Figure 58. For example, the authentication restriction details the type of constraint (*VALUE*), a *description*, *name* and a *non-functional attribute* (*authentication*). These properties are described in all constraints (item 2, figure 58).

Figure 59 shows the service components that must interact with the *application* and the interaction among them. There are four services that are invoked to produce results for the *Publish Music* application: *Spotify, Facebook, Twitter* and *Bank*.

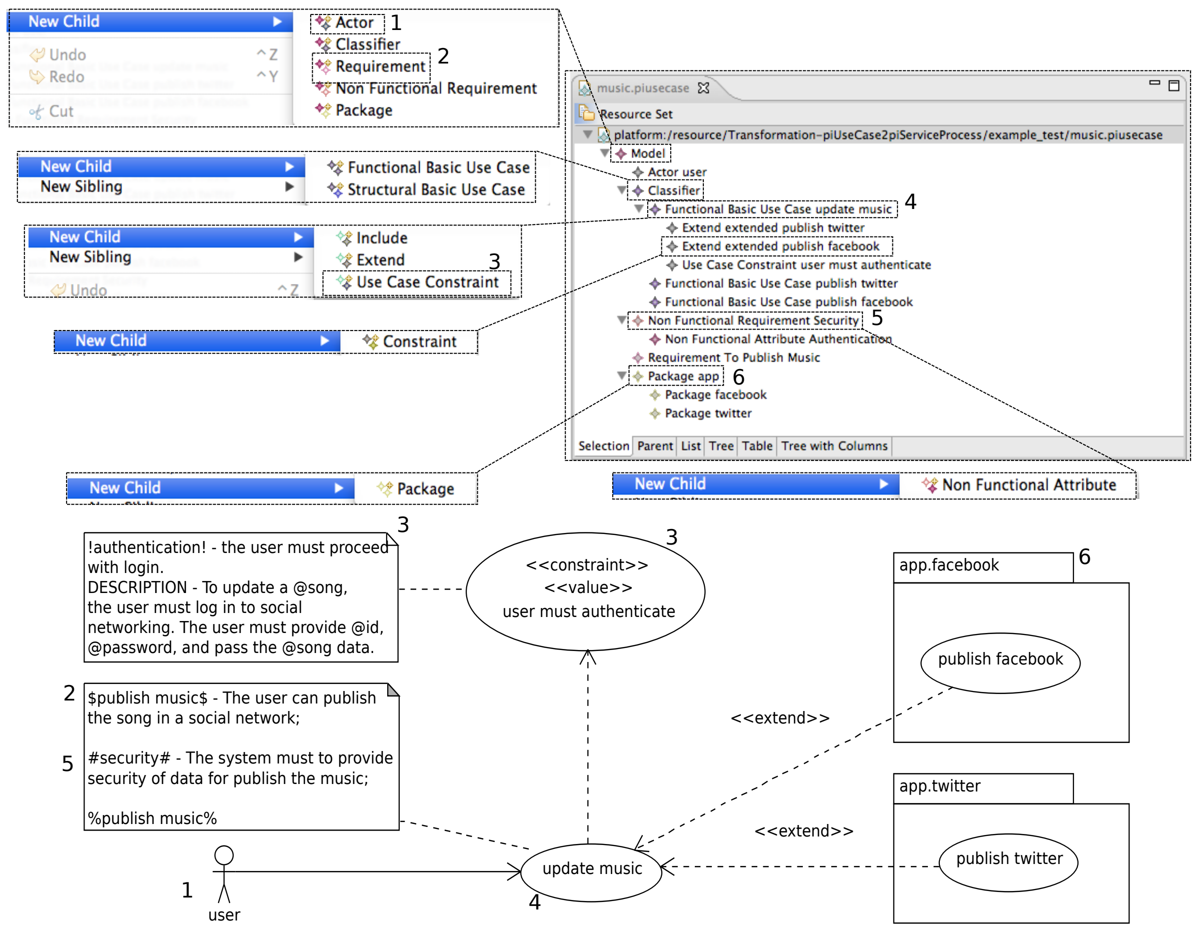
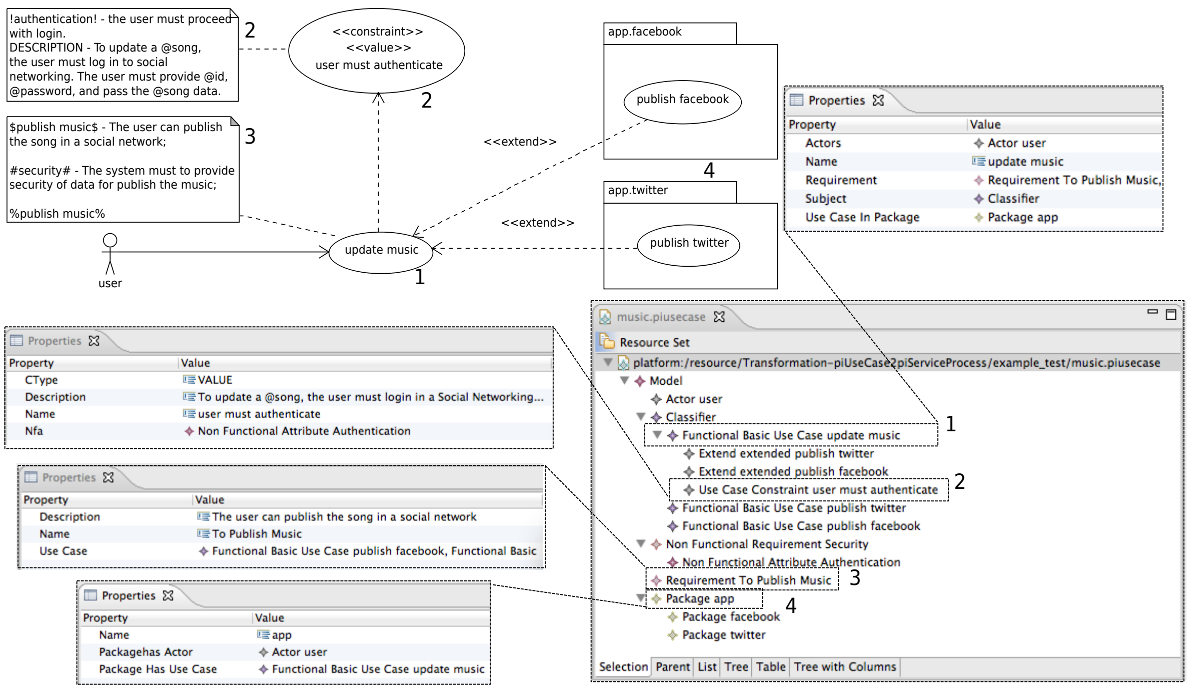

New Child
Undo
Redo
Cut
Actor 1
Classifier
Requirement 2
Non Functional Requirement
Package
New Child
New Sibling
Functional Basic Use Case
Structural Basic Use Case
New Child
New Sibling
Undo
Include
Extend
Use Case Constraint 3
New Child
Constraint
New Child
Package
music.piusecase
Resource Set
platform:/resource/Transformation-piUseCase2piServiceProcess/example_test/music.piusecase
Model
Actor user
Classifier
Functional Basic Use Case update music 4
Extend extended publish twitter
Extend extended publish facebook
Use Case Constraint user must authenticate
Functional Basic Use Case publish twitter
Functional Basic Use Case publish facebook
Non Functional Requirement Security 5
Non Functional Attribute Authentication
Requirement To Publish Music
Package app 6
Package facebook
Package twitter
Selection Parent List Tree Table Tree with Columns
New Child
Non Functional Attribute
3
!authentication! - the user must proceed with login.
DESCRIPTION - To update a @song, the user must log in to social networking. The user must provide @id, @password, and pass the @song data.
3
<<constraint>>
<<value>>
user must authenticate
app.facebook 6
publish facebook
<<extend>>
2
$publish music$ - The user can publish the song in a social network;
5
#security# - The system must to provide security of data for publish the music;
%publish music%
app.twitter
<<extend>>
publish twitter
1
user
update music
4

Figure 57: *To Publish Music* $\pi$-UseCase Environment Detail.


!authentication! - the user must proceed with login.
DESCRIPTION - To update a @song, the user must log in to social networking. The user must provide @id, @password, and pass the @song data.
2
<<constraint>>
<<value>>
user must authenticate
2
$publish music$ - The user can publish the song in a social network;
#security# - The system must to provide security of data for publish the music;
%publish music%
3
app.facebook
publish facebook
<<extend>>
4
app.twitter
<<extend>>
publish twitter
user
update music
1
music.piusecase
Resource Set
platform:/resource/Transformation-piUseCase2piServiceProcess/example_test/music.piusecase
Model
Actor user
Classifier
Functional Basic Use Case update music 1
Extend extended publish twitter
Extend extended publish facebook
Use Case Constraint user must authenticate 2
Functional Basic Use Case publish twitter
Functional Basic Use Case publish facebook
Non Functional Requirement Security
Non Functional Attribute Authentication
Requirement To Publish Music 3
Package app 4
Package facebook
Package twitter
Selection Parent List Tree Table Tree with Columns

Properties

| Property | Value |
|---|---|
| Actors | Actor user |
| Name | update music |
| Requirement | Requirement To Publish Music, |
| Subject | Classifier |
| Use Case In Package | Package app |

Properties

| Property | Value |
|---|---|
| CType | VALUE |
| Description | To update a @song, the user must login in a Social Networking... |
| Name | user must authenticate |
| Nfa | Non Functional Attribute Authentication |

Properties

| Property | Value |
|---|---|
| Description | The user can publish the song in a social network |
| Name | To Publish Music |
| Use Case | Functional Basic Use Case publish facebook, Functional Basic |

Properties

| Property | Value |
|---|---|
| Name | app |
| Packagehas Actor | Actor user |
| Package Has Use Case | Functional Basic Use Case update music |

Figure 58: *To Publish Music*$\pi$-UseCase Environment Detail.

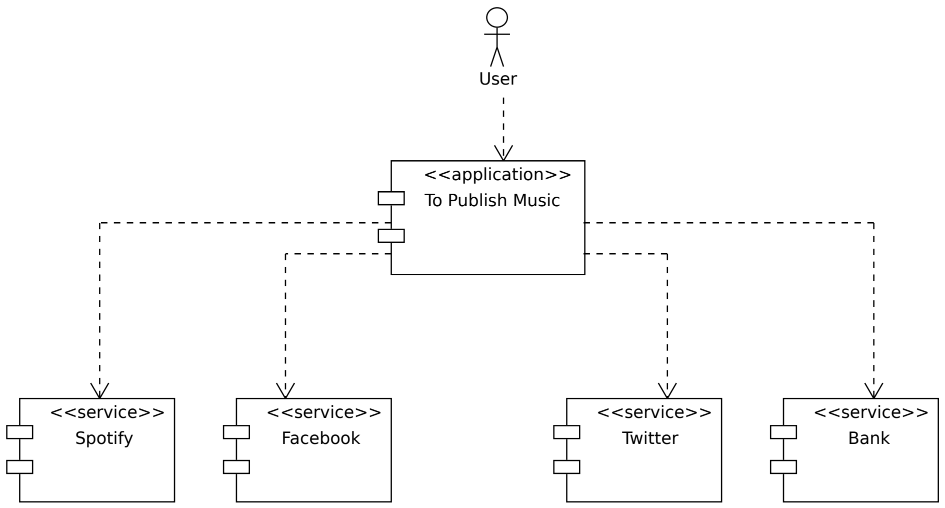


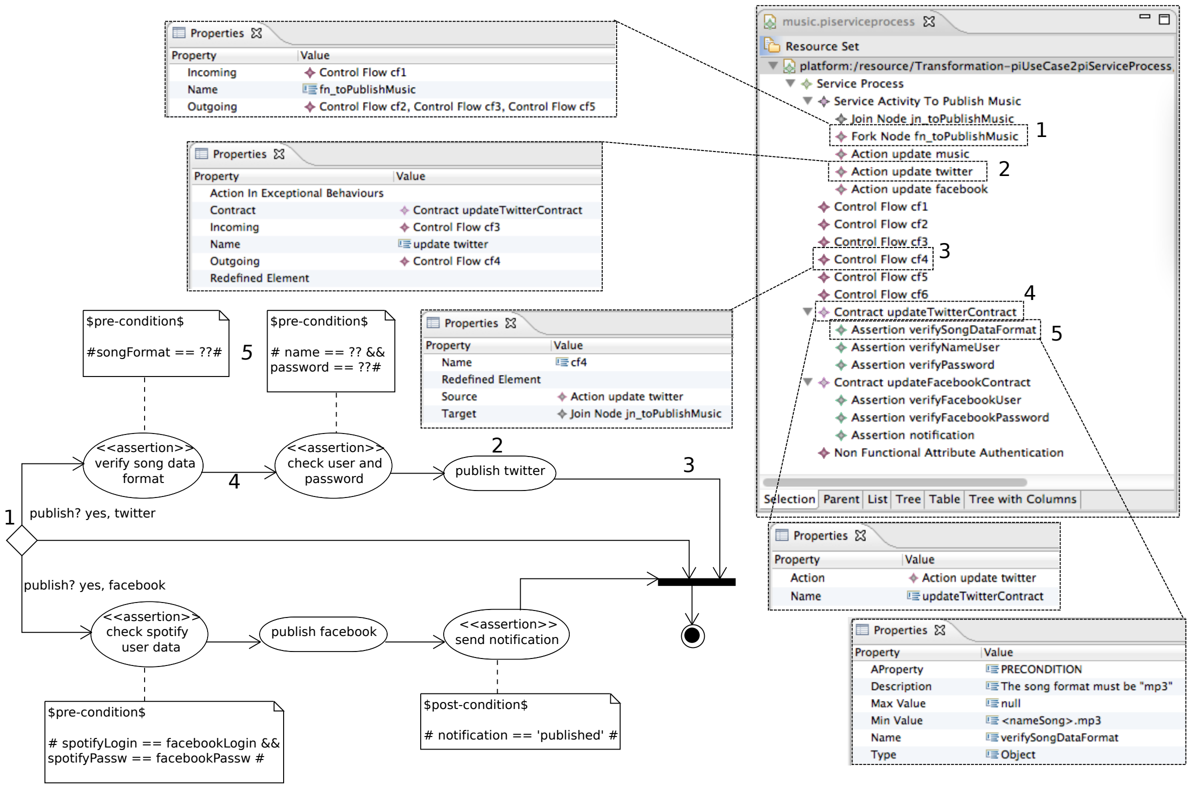
Figure 59: To Publish Music Services.

## 5.1.2 $\pi$-ServiceProcess Model

Figure 60 shows the $\pi$-*ServiceProcess* diagram for the execution of activitiesbased on contracts. We present here the part of the diagram produced from the concepts of the figure 58. The complete $\pi$-*ServiceProcess* diagram of the application is shown in figure 26. Notice that it is necessary to represent each CONSTRAINT modelled in a CONTRACT based service activity diagram. Each constraint described in the previous model is transformed into a contract in the $\pi$-ServiceProcess model.

*Publish Twitter* and *publish Facebook* (figure 60) have pre- and post-conditions assertions that are grouped into a contract for each service: (i) verify if the user data are correct; (ii) if the user is already logged in Spotify; As post-condition, it ensures the complete transaction and verifies whether the payment authorization notification was sent to the user and to Spotify. There may be different restrictions depending on the services participating in the publish process. The user must be logged in Spotify in order to access Facebook, and Facebook must send a "successful login" acknowledge which is verified by the post-condition. For publishing a message, Twitter imposes pre-conditions on the length of the message and user data login. In contrast, Facebook only sends a confirmation acknowledge. Items 4 and 5 in figure 60 show the pre-conditions for "*publish Twitter*" concerning *(i)* the format of the music file and (*ii*) access authorization credentials, namely login and password.

The development of the $\pi$-ServiceProcess refine the case study constraints into contracts for the services as shown in Figure 60.



Resource Set
- platform:/resource/Transformation-piUseCase2piServiceProcess
  - Service Process
    - Service Activity To Publish Music
      - Join Node jn_toPublishMusic
      - Fork Node fn_toPublishMusic 1
      - Action update music
      - Action update twitter 2
      - Action update facebook
    - Control Flow cf1
    - Control Flow cf2
    - Control Flow cf3
    - Control Flow cf4 3
    - Control Flow cf5
    - Control Flow cf6 4
    - Contract updateTwitterContract
      - Assertion verifySongDataFormat 5
      - Assertion verifyNameUser
      - Assertion verifyPassword
    - Contract updateFacebookContract
      - Assertion verifyFacebookUser
      - Assertion verifyFacebookPassword
      - Assertion notification
    - Non Functional Attribute Authentication

Properties

| Property | Value |
|---|---|
| Incoming | Control Flow cf1 |
| Name | fn_toPublishMusic |
| Outgoing | Control Flow cf2, Control Flow cf3, Control Flow cf5 |

Properties

| Property | Value |
|---|---|
| Action In Exceptional Behaviours | |
| Contract | Contract updateTwitterContract |
| Incoming | Control Flow cf3 |
| Name | update twitter |
| Outgoing | Control Flow cf4 |
| Redefined Element | |

Properties

| Property | Value |
|---|---|
| Name | cf4 |
| Redefined Element | |
| Source | Action update twitter |
| Target | Join Node jn_toPublishMusic |

Properties

| Property | Value |
|---|---|
| Action | Action update twitter |
| Name | updateTwitterContract |

Properties

| Property | Value |
|---|---|
| AProperty | PRECONDITION |
| Description | The song format must be "mp3" |
| Max Value | null |
| Min Value | <nameSong>.mp3 |
| Name | verifySongDataFormat |
| Type | Object |

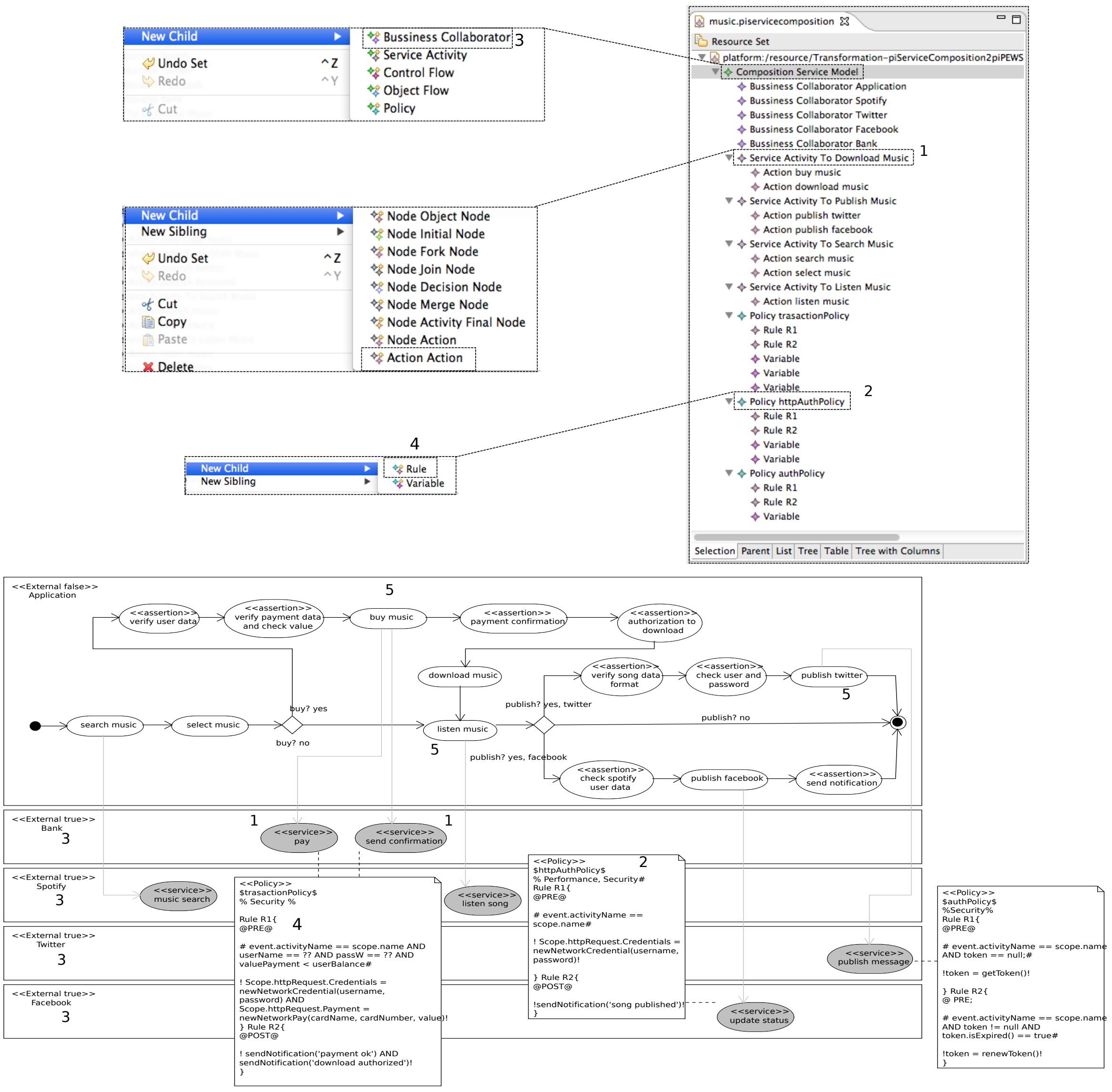
Figure 60: *To Publish Music* $\pi$-ServiceProcess.

## 5.1.3 $\pi$-ServiceComposition Model

There are four external Business Collaborators in this application: *Bank, Spotify, Twitter* and *Facebook* (items 3, figure 61). Figure 61 shows the application services that consist of a set of functions: *search music, select music, buy music, download music, listen music* and *publish music*. The *publish music* action calls two service collaborators (Facebook and Twitter), and the *buy music* action calls two functions of the Bank service collaborator.

The *To Publish Music* $\pi$-ServiceComposition model defines three policies: *transactionPolicy* (item 4, figure 61), *httpAuthPolicy* (item 2) and *authPolicy* (item 6). The *trasactionPolicy* verifies the Bank data restrictions over the payment and confirmation process. The *httpAuthPolicy* policy specifies rules over the Facebook function for the authentication and notification processes. These rules are executed during the publish massage function request. The *authPolicy* policy implements the open authentication protocol required for accessing the Twitter service.

From the *To Publish Music* rules and policies it is possible to model and associate non-functional properties to services' compositions, *e.g.* the integration of information retrieved from different social network services, automatic generation of an integrated view of the op-


New Child
Undo Set ^Z
Redo ^Y
Cut
Bussiness Collaborator
Service Activity
Control Flow
Object Flow
Policy
3
music.piservicecomposition
Resource Set
platform:/resource/Transformation-piServiceComposition2piPEWS
Composition Service Model
Bussiness Collaborator Application
Bussiness Collaborator Spotify
Bussiness Collaborator Twitter
Bussiness Collaborator Facebook
Bussiness Collaborator Bank
Service Activity To Download Music
1
Action buy music
Action download music
Service Activity To Publish Music
Action publish twitter
Action publish facebook
Service Activity To Search Music
Action search music
Action select music
Service Activity To Listen Music
Action listen music
Policy trasactionPolicy
Rule R1
Rule R2
Variable
Variable
Variable
Policy httpAuthPolicy
2
Rule R1
Rule R2
Variable
Variable
Policy authPolicy
Rule R1
Rule R2
Variable
Selection | Parent | List | Tree | Table | Tree with Columns
New Child
New Sibling
Undo Set ^Z
Redo ^Y
Cut
Copy
Paste
Delete
Node Object Node
Node Initial Node
Node Fork Node
Node Join Node
Node Decision Node
Node Merge Node
Node Activity Final Node
Node Action
Action Action
4
New Child
New Sibling
Rule
Variable
<<External false>> Application
5
<<assertion>> verify user data
<<assertion>> verify payment data and check value
buy music
<<assertion>> payment confirmation
<<assertion>> authorization to download
download music
<<assertion>> verify song data format
<<assertion>> check user and password
publish twitter
5
search music
select music
buy? yes
buy? no
listen music
5
publish? yes, twitter
publish? no
publish? yes, facebook
<<assertion>> check spotify user data
publish facebook
<<assertion>> send notification
<<External true>> Bank
3
1
<<service>> pay
1
<<service>> send confirmation
<<External true>> Spotify
3
<<service>> music search
<<service>> listen song
<<External true>> Twitter
3
<<service>> publish message
<<External true>> Facebook
3
<<service>> update status
4
2

```
<<Policy>>
$trasactionPolicy$
% Security %

Rule R1{
@PRE@

# event.activityName == scope.name AND
userName == ?? AND passW == ?? AND
valuePayment < userBalance#

! Scope.httpRequest.Credentials =
newNetworkCredential(username,
password) AND
Scope.httpRequest.Payment =
newNetworkPay(cardName, cardNumber, value)!
} Rule R2{
@POST@

! sendNotification('payment ok') AND
sendNotification('download authorized')!
}
```

```
<<Policy>>
$httpAuthPolicy$
% Performance, Security#
Rule R1{
@PRE@

# event.activityName ==
scope.name#

! Scope.httpRequest.Credentials =
newNetworkCredential(username,
password)!

} Rule R2{
@POST@

!sendNotification('song published')!
}
```

```
<<Policy>>
$authPolicy$
%Security%
Rule R1{
@PRE@

# event.activityName == scope.name
AND token == null;#

!token = getToken()!

} Rule R2{
@ PRE;

# event.activityName == scope.name
AND token != null AND
token.isExpired() == true#

!token = renewToken()!
}
```

Figure 61: *To Publish Music* $\pi$-ServiceComposition Model.

erations executed in different social networks or for providing security in the communication channel when the payment service is called.

From this stage, the $\pi$-*PEWS* model (PSM) depicted in figures 32 and 33 can be generated.

## 5.2 Case Study 2: Crime Map

The *Crime Map* case study addresses the design and implementation of a system that deals with statistical data on criminal activities. The system works in conjunction with third-party services, in order to process a search supplied by a user. The system must provide a public utility service to inform citizens about the real crime situation in a particular location or region.

Making use of third party services, the application must provide a statistics crime portal. The result depends on user queries and upon a user demand results can be presented on a map. The system also interacts with micro-blogging, for looking up messages containing specific crime information. The presentation of the data concerning criminal activities is provided by the *Police service*. The presentation of a locality map crime is made by the *Google Maps service* and messages presentation on the micro-blogging is done by the *Twitter service*. The system also has a *Post service*, for searching an address location that can be then presented in a map by Google maps. The scenarios to query data and crime statistics have the following actors: *(i) User, (ii) Map service, (iii) Post service, (iv) Micro-blogging service* and *(v) Police service*, to query the criminal activities (figure 62).

Considering the application functions, the crimes may be searched by types, *e.g. theft, murder, kidnapping*, etc. The search can also be made by city' district or region, for example, *center, west, south, north* or *east*, and it can process search temporal criteria, for example, crimes of the previous day of the last weeks or months. Finally, the service can also process conjunctive queries, *e.g.*, *the number of kidnapping of a specific district in the last month.*

Whenever a new crime is published by the police, the system should post a message on Twitter stating the new crime, with a specific hashtag, *e.g. #crime043_kidnapping_center_05-08-2012*. The hashtag structure contains the crime number, type, local and the day it happens. From this, users can comment about the crime and these comment will appear in the application interface. All users must have an account to use the application.

The application functionalities are: *Choose search type of crime; see crime information; Receive information on the chosen query; See map; List crimes;* and *Comment about a crime.*

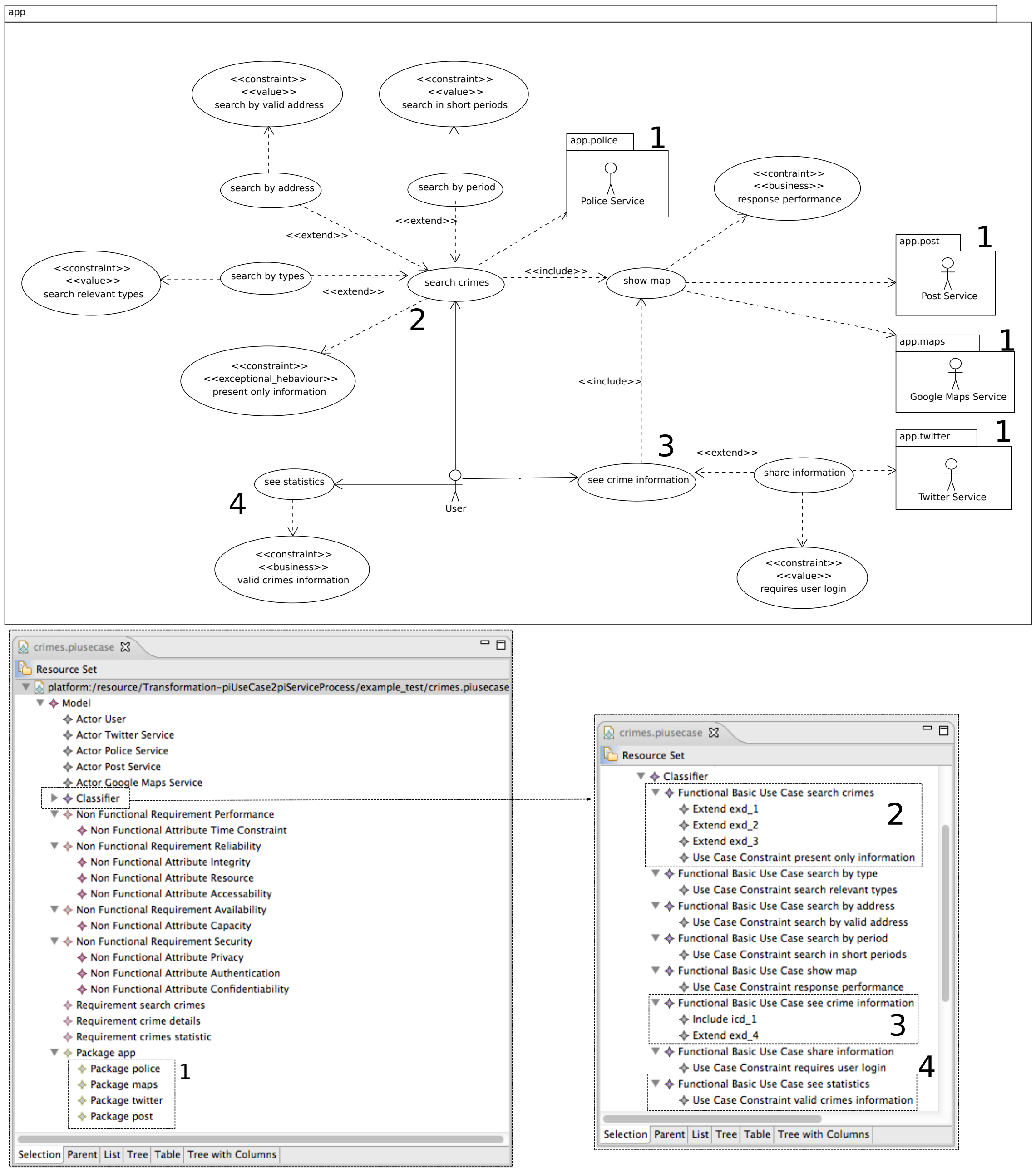
Non-functional requirements for this application include: *queries response performance, crime information validation*, *crime information presented in a map* and *ensuring crimes data security.*

### 5.2.1 $\pi$-UseCase Model

Figure 62 presents the Crime Map $\pi$-UseCase model. This model includes examples of *business, value* and *exceptional behaviours* constraints. The application accesses data from third party services to provide the user an easy way to visualize new information, these services are: *Map, Police, Postal* and *Microb-logging Services*.

The user interacts with the application that will perform the services for obtain and publish details about the crimes. The services are modeled as actors (they are external applications).

Each service has a set of functions that can be performed. The $\pi$-UseCase model describes both the external services and the functionality of the application (see figure 62).


app
<<constraint>> <<value>> search by valid address
<<constraint>> <<value>> search in short periods
app.police
1
Police Service
<<contraint>> <<business>> response performance
search by address
search by period
<<extend>>
<<extend>>
app.post
1
Post Service
<<constraint>> <<value>> search relevant types
search by types
<<extend>>
search crimes
<<include>>
show map
2
app.maps
1
Google Maps Service
<<constraint>> <<exceptional_hebaviour>> present only information
<<include>>
app.twitter
1
Twitter Service
3
<<extend>>
see statistics
4
User
see crime information
share information
<<constraint>> <<business>> valid crimes information
<<constraint>> <<value>> requires user login
1
2
3
4

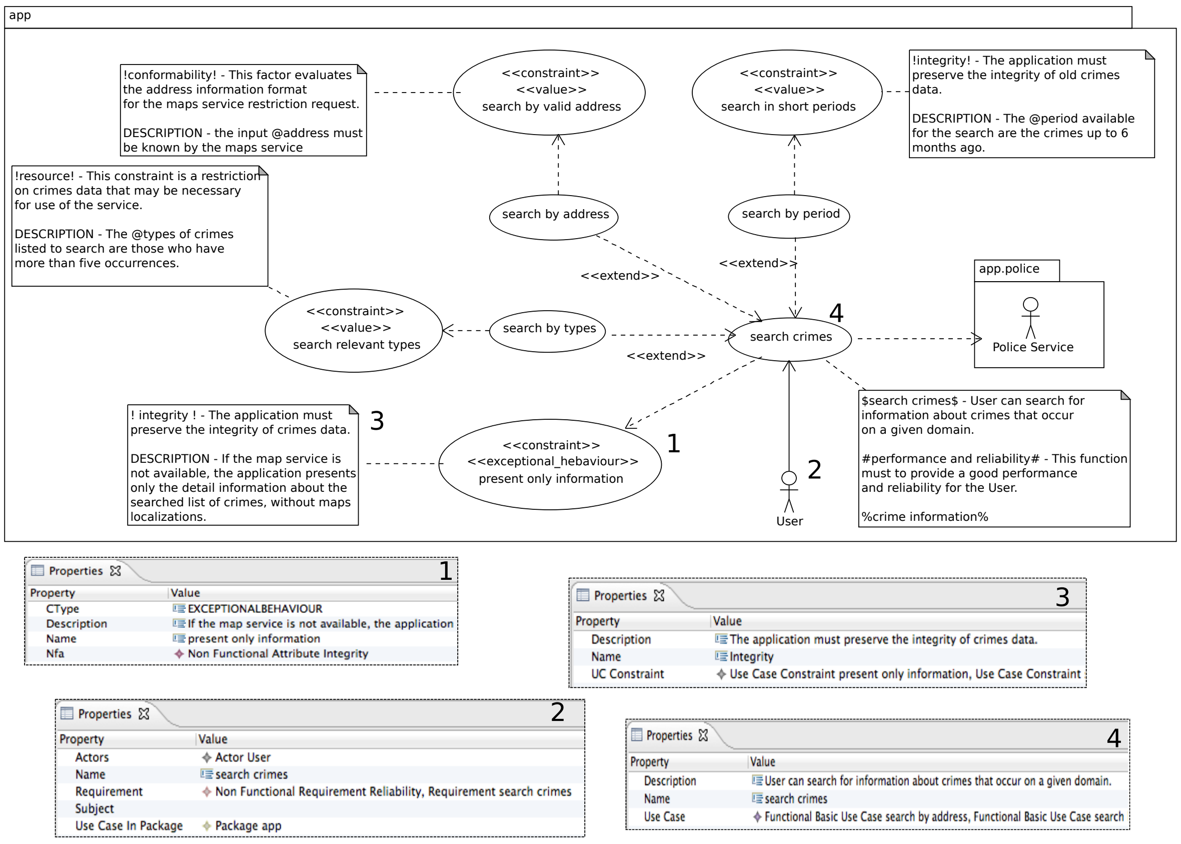
```
crimes.piusecase
Resource Set
▼ platform:/resource/Transformation-piUseCase2piServiceProcess/example_test/crimes.piusecase
  ▼ Model
      Actor User
      Actor Twitter Service
      Actor Police Service
      Actor Post Service
      Actor Google Maps Service
    ▶ Classifier
    ▼ Non Functional Requirement Performance
        Non Functional Attribute Time Constraint
    ▼ Non Functional Requirement Reliability
        Non Functional Attribute Integrity
        Non Functional Attribute Resource
        Non Functional Attribute Accessability
    ▼ Non Functional Requirement Availability
        Non Functional Attribute Capacity
    ▼ Non Functional Requirement Security
        Non Functional Attribute Privacy
        Non Functional Attribute Authentication
        Non Functional Attribute Confidentiability
      Requirement search crimes
      Requirement crime details
      Requirement crimes statistic
    ▼ Package app
        Package police
        Package maps
        Package twitter
        Package post
Selection | Parent | List | Tree | Table | Tree with Columns
```

```
crimes.piusecase
Resource Set
  ▼ Classifier
    ▼ Functional Basic Use Case search crimes
        Extend exd_1
        Extend exd_2
        Extend exd_3
        Use Case Constraint present only information
    ▼ Functional Basic Use Case search by type
        Use Case Constraint search relevant types
    ▼ Functional Basic Use Case search by address
        Use Case Constraint search by valid address
    ▼ Functional Basic Use Case search by period
        Use Case Constraint search in short periods
    ▼ Functional Basic Use Case show map
        Use Case Constraint response performance
    ▼ Functional Basic Use Case see crime information
        Include icd_1
        Extend exd_4
    ▼ Functional Basic Use Case share information
        Use Case Constraint requires user login
    ▼ Functional Basic Use Case see statistics
        Use Case Constraint valid crimes information
Selection | Parent | List | Tree | Table | Tree with Columns
```

Figure 62: Crime Map $\pi$-UseCase.

The user can search for crimes and view its occurrences in a map. The search may be performed by *type of crime, region* or *period*. The user can also perform a search that combines these criteria. If the application's user does not choose any option, the search will retrieve up to 30 recent occurrences. If the location maps service is unavailable or there is no response from it after 5 seconds, only the crimes information will be presented, without a map view.

Figure 62 also presents the $\pi$-UseCase model tree representation generated by the $\pi$SOD-M environment model .Figures 63 and 64 detail the use cases and constraints presented in Figure 62.



!conformability! - This factor evaluates the address information format for the maps service restriction request.

DESCRIPTION - the input @address must be known by the maps service

!integrity! - The application must preserve the integrity of old crimes data.

DESCRIPTION - The @period available for the search are the crimes up to 6 months ago.

!resource! - This constraint is a restriction on crimes data that may be necessary for use of the service.

DESCRIPTION - The @types of crimes listed to search are those who have more than five occurrences.

! integrity ! - The application must preserve the integrity of crimes data.

DESCRIPTION - If the map service is not available, the application presents only the detail information about the searched list of crimes, without maps localizations.

$search crimes$ - User can search for information about crimes that occur on a given domain.

#performance and reliability# - This function must to provide a good performance and reliability for the User.

%crime information%

Properties 1

| Property | Value |
|---|---|
| CType | EXCEPTIONALBEHAVIOUR |
| Description | If the map service is not available, the application |
| Name | present only information |
| Nfa | Non Functional Attribute Integrity |

Properties 3

| Property | Value |
|---|---|
| Description | The application must preserve the integrity of crimes data. |
| Name | Integrity |
| UC Constraint | Use Case Constraint present only information, Use Case Constraint |

Properties 2

| Property | Value |
|---|---|
| Actors | Actor User |
| Name | search crimes |
| Requirement | Non Functional Requirement Reliability, Requirement search crimes |
| Subject | |
| Use Case In Package | Package app |

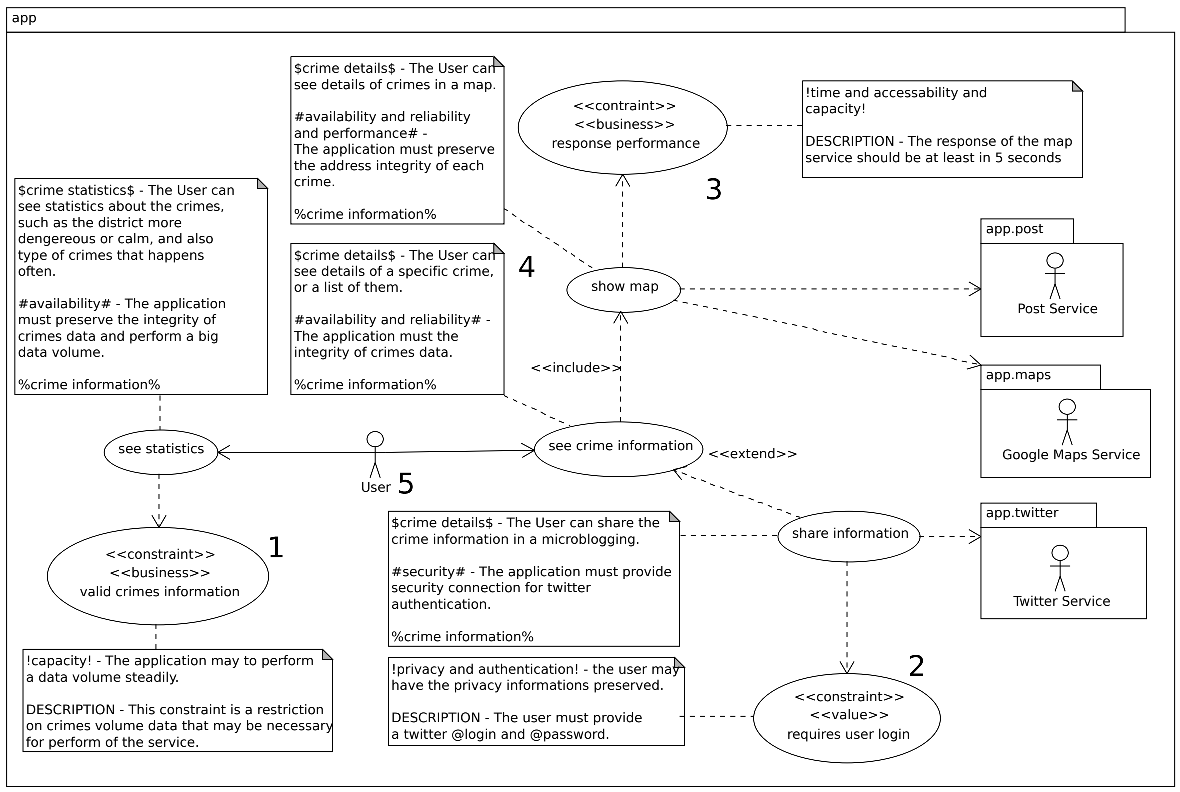
Properties 4

| Property | Value |
|---|---|
| Description | User can search for information about crimes that occur on a given domain. |
| Name | search crimes |
| Use Case | Functional Basic Use Case search by address, Functional Basic Use Case search |

Figure 63: *Search crime* $\pi$-UseCase Detail.

The *search crimes* use case (Figure 63) requests the *Police* service to verify the crimes according to the parameters supplied by the user. The restriction of this use case depends on the parameters selected by the user. If the user chooses one or several types of specific crime's type, the system will present those crimes that have more than five occurrences. For *search by address*, the address must comply with the restrictions of the police service, the research can be done by street, neighborhood, city or district. *Search by period* will provide only occurrences within the last six months.

Figure 64 describes the design of the use cases: *see crime information, see statistics, show map* and *share information*. To share information of a crime, it is necessary to access the Twitter micro-blogging service to proceed with the authentication and posting. The service requires the user id and password, however the application must ensure the User privacy information while performing authentication. To see statistics of crimes, the application should process a larger volume of data to generate statistical results. Thus, it is required to provide resources to ensure



$crime details$ - The User can see details of crimes in a map.

#availability and reliability and performance# - The application must preserve the address integrity of each crime.

%crime information%

!time and accessability and capacity!

DESCRIPTION - The response of the map service should be at least in 5 seconds

$crime statistics$ - The User can see statistics about the crimes, such as the district more dengereous or calm, and also type of crimes that happens often.

#availability# - The application must preserve the integrity of crimes data and perform a big data volume.

%crime information%

$crime details$ - The User can see details of a specific crime, or a list of them.

#availability and reliability# - The application must the integrity of crimes data.

%crime information%

$crime details$ - The User can share the crime information in a microblogging.

#security# - The application must provide security connection for twitter authentication.

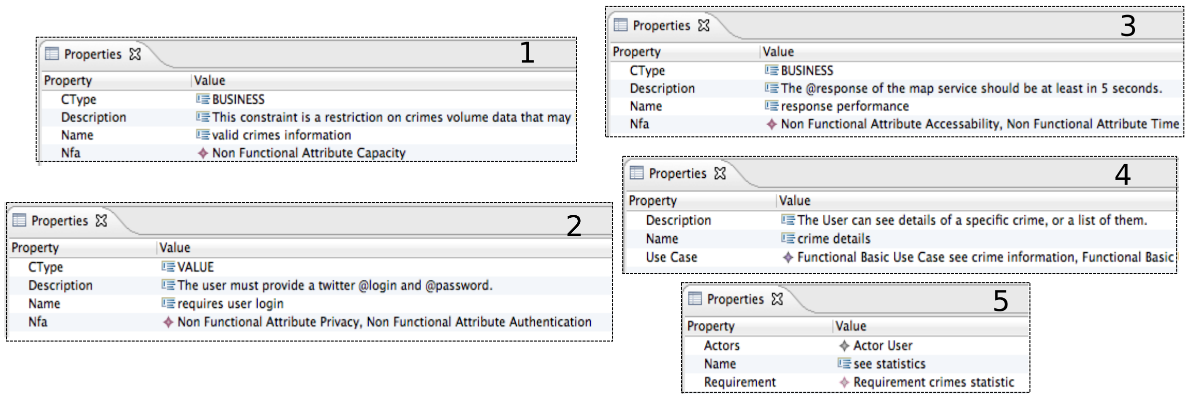
%crime information%

!capacity! - The application may to perform a data volume steadily.

DESCRIPTION - This constraint is a restriction on crimes volume data that may be necessary for perform of the service.

!privacy and authentication! - the user may have the privacy informations preserved.

DESCRIPTION - The user must provide a twitter @login and @password.



Properties 1

| Property | Value |
|---|---|
| CType | BUSINESS |
| Description | This constraint is a restriction on crimes volume data that may |
| Name | valid crimes information |
| Nfa | Non Functional Attribute Capacity |

Properties 2

| Property | Value |
|---|---|
| CType | VALUE |
| Description | The user must provide a twitter @login and @password. |
| Name | requires user login |
| Nfa | Non Functional Attribute Privacy, Non Functional Attribute Authentication |

Properties 3

| Property | Value |
|---|---|
| CType | BUSINESS |
| Description | The @response of the map service should be at least in 5 seconds. |
| Name | response performance |
| Nfa | Non Functional Attribute Accessability, Non Functional Attribute Time |

Properties 4

| Property | Value |
|---|---|
| Description | The User can see details of a specific crime, or a list of them. |
| Name | crime details |
| Use Case | Functional Basic Use Case see crime information, Functional Basic |

Properties 5

| Property | Value |
|---|---|
| Actors | Actor User |
| Name | see statistics |
| Requirement | Requirement crimes statistic |

Figure 64: *See crime information* $\pi$-UseCase Detail.

the data processing performance. The restriction on map presentation is over response time and accessibility of the *Postal* and *Google maps* services. The response time limit is 5 seconds. If the map is not processed, only the information of the crime are presented. Figures 63 and 64 also show the properties used in the $\pi$SOD-M environment and the equivalent concepts in the model.

After the identification the various functions that are required by the system to perform the business services, the $\pi$-ServiceProcess model is used to represent the workflow necessary to perform a service within the system.

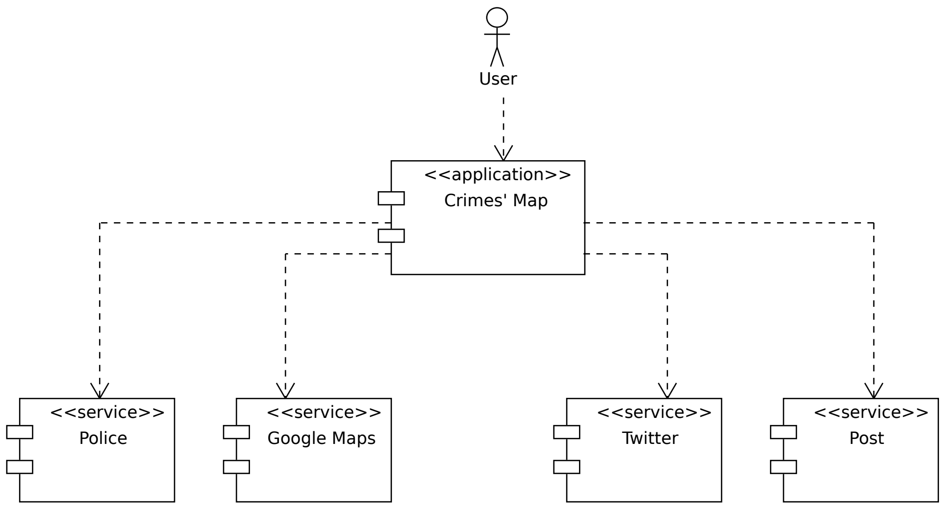


Figure 65: Crime Map Services.

### 5.2.2 $\pi$-ServiceProcess Model

The $\pi$-ServiceProcess model is used to represent the workflow necessary to perform the system service. The service processes execution is described in two steps. Figures 66 and 68 show the flow of functions execution described in the $\pi$-UseCase models shown in Figures 63 and 64. Each diagram can represent different business services (see figures 63 and 64).

The diagrams of figures 66 and 68 were obtained by applying the *$\pi$-UseCase2$\pi$-ServiceProcess* transformation to the diagrams of figures 63 and 64, respectively.

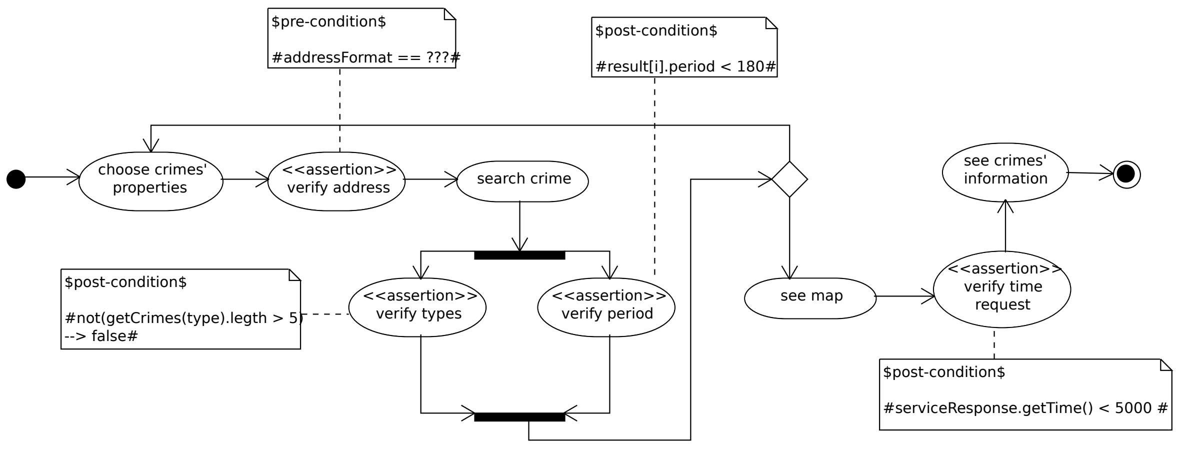
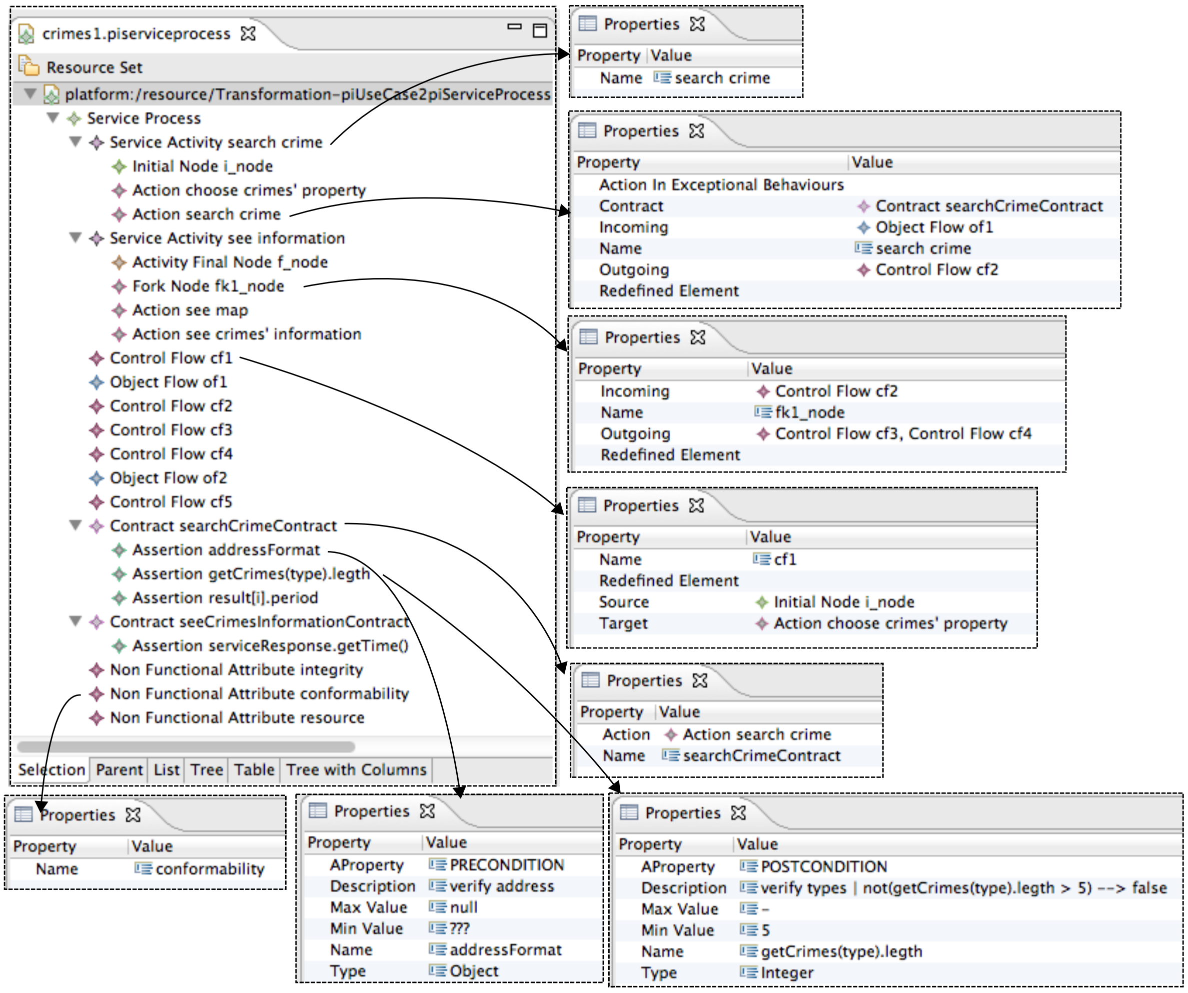

$pre-condition$
#addressFormat == ???#
$post-condition$
#result[i].period < 180#
choose crimes' properties
<<assertion>> verify address
search crime
$post-condition$
#not(getCrimes(type).legth > 5 --> false#
<<assertion>> verify types
<<assertion>> verify period
see map
<<assertion>> verify time request
see crimes' information
$post-condition$
#serviceResponse.getTime() < 5000 #

Figure 66: *Crime information* $\pi$-ServiceProcess.

Each constraint of figures 66 and 68 are transformed into assertions. For example, the assertions of address format, crime types and time are restrictions over the action *search crime*, being one pre-condition and two post-conditions. These restrictions form the contract on the



crimes1.piserviceprocess

Resource Set
- platform:/resource/Transformation-piUseCase2piServiceProcess
  - Service Process
    - Service Activity search crime
      - Initial Node i_node
      - Action choose crimes' property
      - Action search crime
    - Service Activity see information
      - Activity Final Node f_node
      - Fork Node fk1_node
      - Action see map
      - Action see crimes' information
    - Control Flow cf1
    - Object Flow of1
    - Control Flow cf2
    - Control Flow cf3
    - Control Flow cf4
    - Object Flow of2
    - Control Flow cf5
    - Contract searchCrimeContract
      - Assertion addressFormat
      - Assertion getCrimes(type).legth
      - Assertion result[i].period
    - Contract seeCrimesInformationContract
      - Assertion serviceResponse.getTime()
    - Non Functional Attribute integrity
    - Non Functional Attribute conformability
    - Non Functional Attribute resource

Selection | Parent | List | Tree | Table | Tree with Columns

Properties

| Property | Value |
|---|---|
| Name | search crime |

Properties

| Property | Value |
|---|---|
| Action In Exceptional Behaviours | |
| Contract | Contract searchCrimeContract |
| Incoming | Object Flow of1 |
| Name | search crime |
| Outgoing | Control Flow cf2 |
| Redefined Element | |

Properties

| Property | Value |
|---|---|
| Incoming | Control Flow cf2 |
| Name | fk1_node |
| Outgoing | Control Flow cf3, Control Flow cf4 |
| Redefined Element | |

Properties

| Property | Value |
|---|---|
| Name | cf1 |
| Redefined Element | |
| Source | Initial Node i_node |
| Target | Action choose crimes' property |

Properties

| Property | Value |
|---|---|
| Action | Action search crime |
| Name | searchCrimeContract |

Properties

| Property | Value |
|---|---|
| Name | conformability |

Properties

| Property | Value |
|---|---|
| AProperty | PRECONDITION |
| Description | verify address |
| Max Value | null |
| Min Value | ??? |
| Name | addressFormat |
| Type | Object |

Properties

| Property | Value |
|---|---|
| AProperty | POSTCONDITION |
| Description | verify types \| not(getCrimes(type).legth > 5) --> false |
| Max Value | - |
| Min Value | 5 |
| Name | getCrimes(type).legth |
| Type | Integer |

Figure 67: *Crime information* $\pi$-ServiceProcess Environment.

*search crime* function. Similarly, for viewing maps, if the answer to the request takes longer than 5 seconds (5000 milliseconds), the map display is suspended and only the information is presented. Each assertion in this $\pi$-ServiceProcess model stems from the $\pi$-UseCase model constraints. These restrictions are shown in Figure 66. Figure 67 presents the equivalent model generated by the methodology transformation in the $\pi$SOD-M environment for the process detailed in Figure 66. Each node defines the process element and its properties.

In Figure 67, *i_node* corresponds to the initial process node. ACTIONS are grouped into SERVICE ACTIVITIES, such as *choose crimes property* action that is part of the *search crime* service activity. A CONTROL FLOW is an edge that links two nodes, for example *cf1* connect the initial node *i_node* (source) with the *choose crimes property* action (target). Figure 67 also presents the assertions described in Figure 66 grouped into CONTRACTS. The contracts are: *searchCrimeContract* and *seeCrimeInformationContract*. These contracts are related with the actions *search crime* and *see crimes information*.

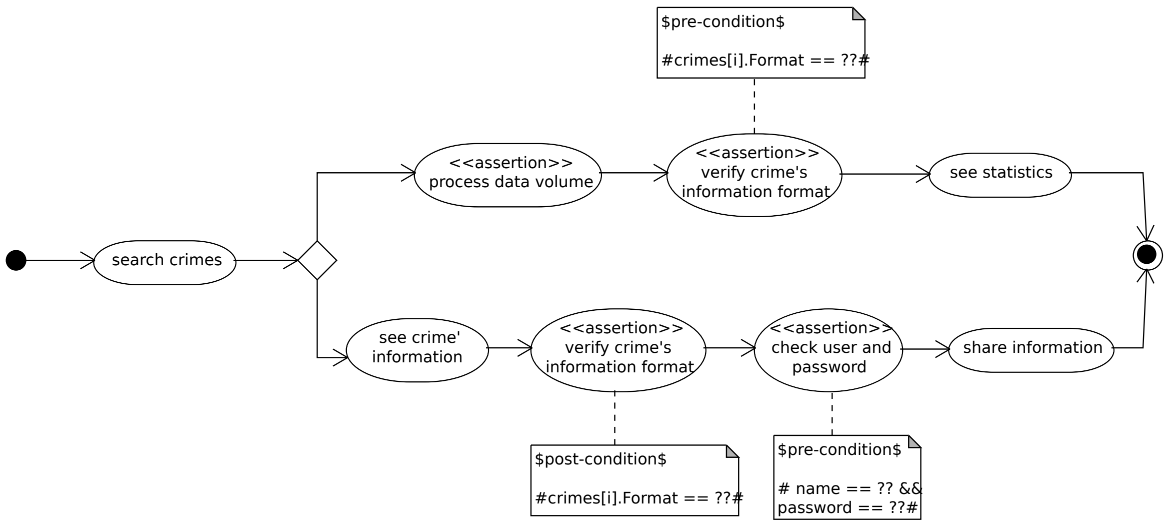

$pre-condition$
#crimes[i].Format == ??#
<<assertion>> process data volume
<<assertion>> verify crime's information format
see statistics
search crimes
see crime' information
<<assertion>> verify crime's information format
<<assertion>> check user and password
share information
$post-condition$
#crimes[i].Format == ??#
$pre-condition$
# name == ?? && password == ??#

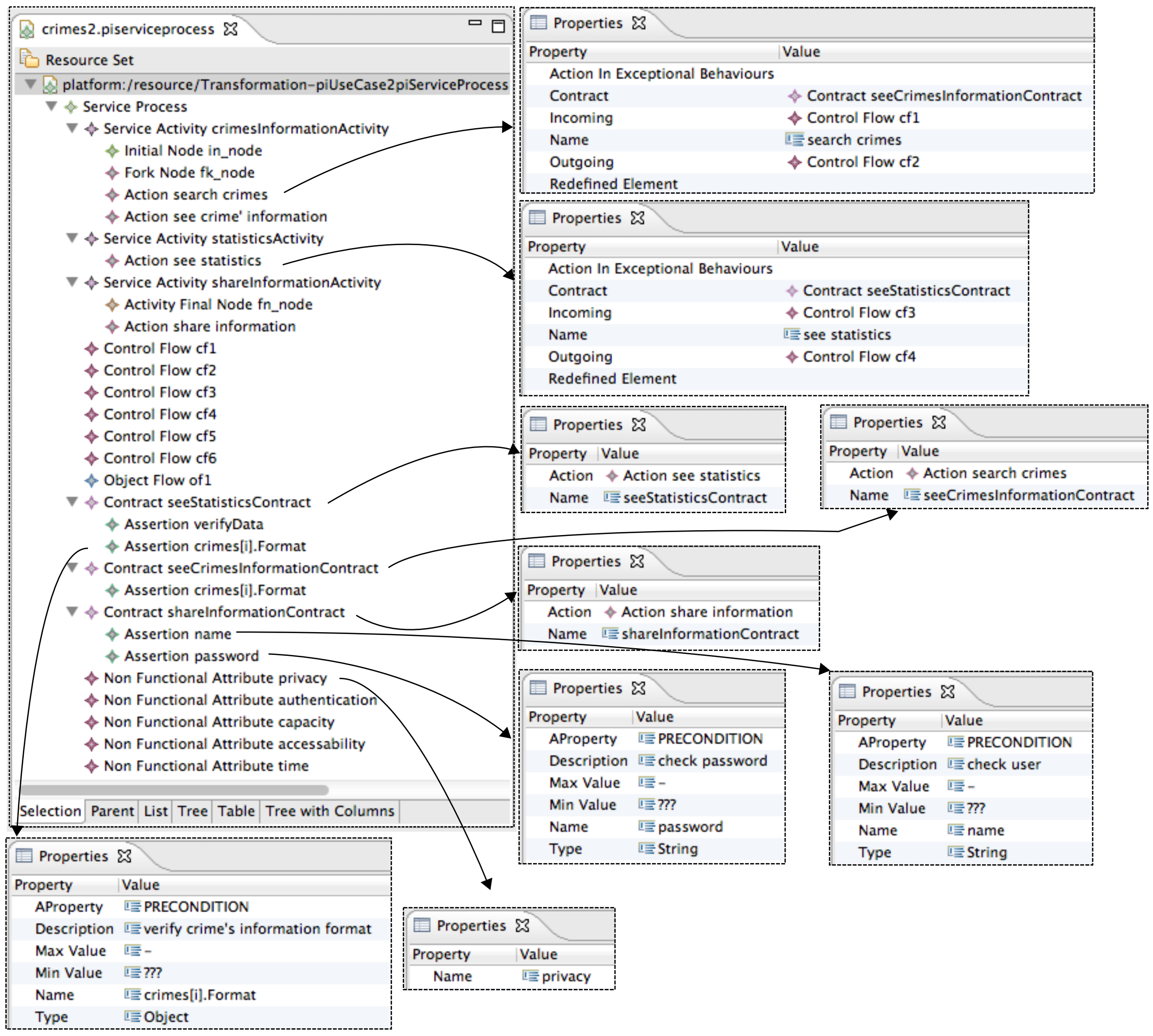
Figure 68: *See crime statistic and share information* $\pi$-ServiceProcess Detail.

Figure 68 presents the assertions over the *see statistics* and *share information* actions. Both have 2 pre-conditions. To *share information* on Twitter it is necessary to verify the format of crime information (140 characters) and the Twitter id and password, for authentication. Regarding the *see statistics* action, there is a business restriction over data volume, and a value restriction over the crime information format. Thus, the presentation of the crimes statistics must be done after the contract verification. Figure 69 presents the equivalent model generated by the methodology transformation in the $\pi$SOD-M environment for the process detailed in Figure 68. Each node defines the process element and its properties. Figure 69 also presents the assertions grouped into three contracts that are related with its specific actions. The contracts are: *seeStatisticsContract*, *seeCrimeInformationContract* and *shareInformationContract*. The contracts are related with the action *see statistics*, *see crimes information* and *share information*, respectively.

These models provide an overview of the business processes from the requirements described in the $\pi$-UseCase model. These models also provide a more detailed view of the execution of business processes and the possible interaction with external services. The description of this interaction is the result of the refinement of these models ($\pi$-ServiceProcess models) through the $\pi$-ServiceComposition model.

### 5.2.3 $\pi$-ServiceComposition Model

Figures 70 and 72 show the $\pi$-ServiceComposition models, which represent the service composition processes of each business service and additional indication of which are members



```
crimes2.piserviceprocess
Resource Set
▼ platform:/resource/Transformation-piUseCase2piServiceProcess
  ▼ Service Process
    ▼ Service Activity crimesInformationActivity
        Initial Node in_node
        Fork Node fk_node
        Action search crimes
        Action see crime' information
    ▼ Service Activity statisticsActivity
        Action see statistics
    ▼ Service Activity shareInformationActivity
        Activity Final Node fn_node
        Action share information
      Control Flow cf1
      Control Flow cf2
      Control Flow cf3
      Control Flow cf4
      Control Flow cf5
      Control Flow cf6
      Object Flow of1
    ▼ Contract seeStatisticsContract
        Assertion verifyData
        Assertion crimes[i].Format
    ▼ Contract seeCrimesInformationContract
        Assertion crimes[i].Format
    ▼ Contract shareInformationContract
        Assertion name
        Assertion password
      Non Functional Attribute privacy
      Non Functional Attribute authentication
      Non Functional Attribute capacity
      Non Functional Attribute accessability
      Non Functional Attribute time
Selection | Parent | List | Tree | Table | Tree with Columns
```

Properties

| Property | Value |
|---|---|
| Action In Exceptional Behaviours | |
| Contract | Contract seeCrimesInformationContract |
| Incoming | Control Flow cf1 |
| Name | search crimes |
| Outgoing | Control Flow cf2 |
| Redefined Element | |

Properties

| Property | Value |
|---|---|
| Action In Exceptional Behaviours | |
| Contract | Contract seeStatisticsContract |
| Incoming | Control Flow cf3 |
| Name | see statistics |
| Outgoing | Control Flow cf4 |
| Redefined Element | |

Properties

| Property | Value |
|---|---|
| Action | Action see statistics |
| Name | seeStatisticsContract |

Properties

| Property | Value |
|---|---|
| Action | Action search crimes |
| Name | seeCrimesInformationContract |

Properties

| Property | Value |
|---|---|
| Action | Action share information |
| Name | shareInformationContract |

Properties

| Property | Value |
|---|---|
| AProperty | PRECONDITION |
| Description | check password |
| Max Value | - |
| Min Value | ??? |
| Name | password |
| Type | String |

Properties

| Property | Value |
|---|---|
| AProperty | PRECONDITION |
| Description | check user |
| Max Value | - |
| Min Value | ??? |
| Name | name |
| Type | String |

Properties

| Property | Value |
|---|---|
| AProperty | PRECONDITION |
| Description | verify crime's information format |
| Max Value | - |
| Min Value | ??? |
| Name | crimes[i].Format |
| Type | Object |

Properties

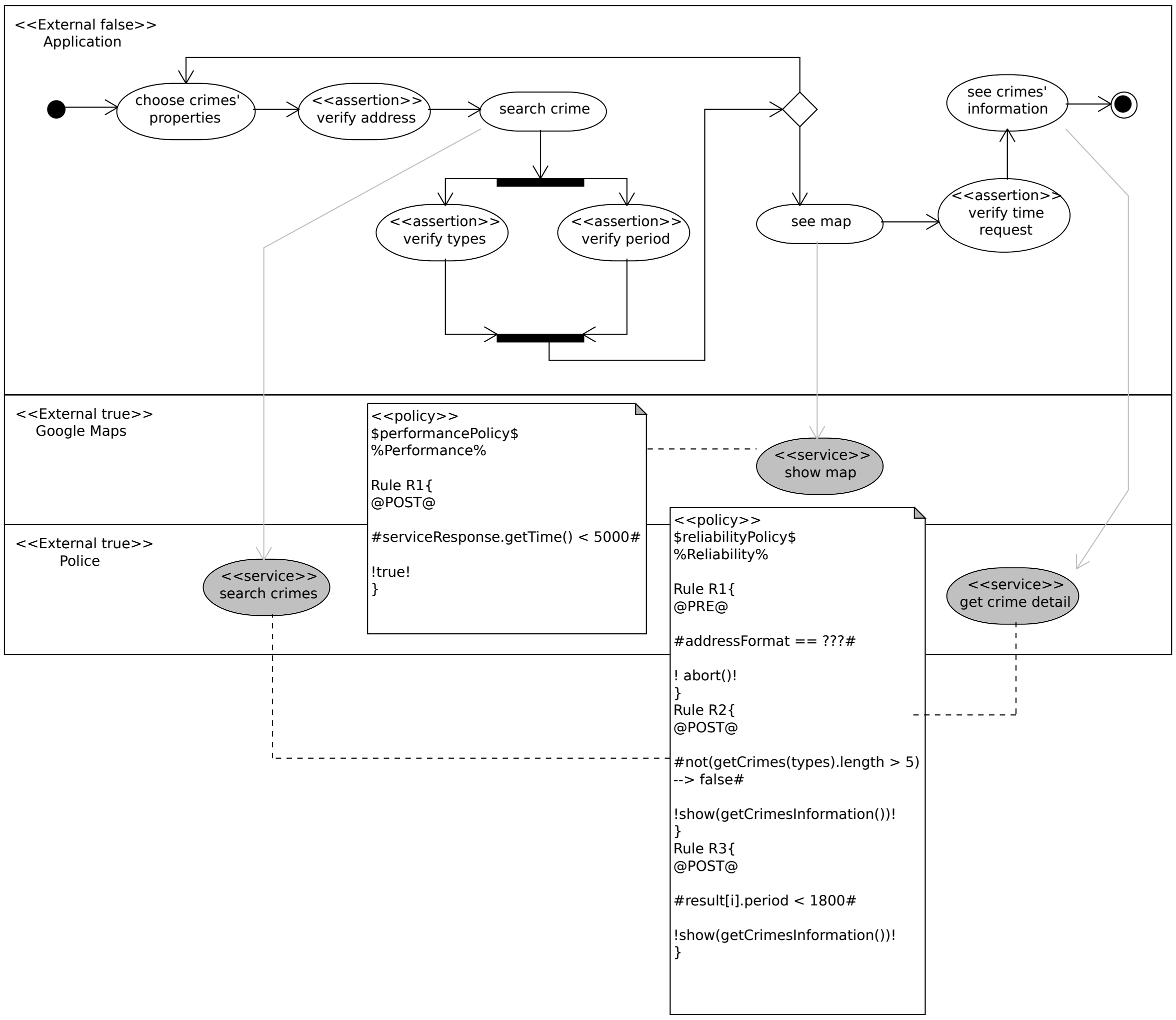
| Property | Value |
|---|---|
| Name | privacy |

Figure 69: *Crime statistic and share information* $\pi$-ServiceComposition Environment.

of the business that execute the action. Actions are derived from service activities, identifying the set of actions that are necessary for the completion of each service.

The main partition expresses the *Application* execution (*External false*) that represents the general process. Actions have the equivalent service relation, which realize the external service, such as *Police*, *Google Maps* or *Twitter* services. They are expressed by a BUSINESS COLLABORATOR. Both, Figure 70 and 72 present two external BUSINESS COLLABORATORS describing the application services that are invoked. These $\pi$-ServiceComposition models refine the $\pi$-ServiceProcess models, matching action with real service functions that may be executed by the system application.

For each $\pi$-ServiceComposition model, the contracts described in the $\pi$-ServiceProcess model are grouped in policies. Figure 70 describes the *performancePolicy* and *reliabilityPolicy* policies, both associated with a service activity and its actions. It is important to highlight that


<<External false>>
Application
choose crimes' properties
<<assertion>> verify address
search crime
<<assertion>> verify types
<<assertion>> verify period
see map
<<assertion>> verify time request
see crimes' information
<<External true>>
Google Maps
<<service>> show map
<<External true>>
Police
<<service>> search crimes
<<service>> get crime detail

```
<<policy>>
$performancePolicy$
%Performance%

Rule R1{
@POST@

#serviceResponse.getTime() < 5000#

!true!
}
```

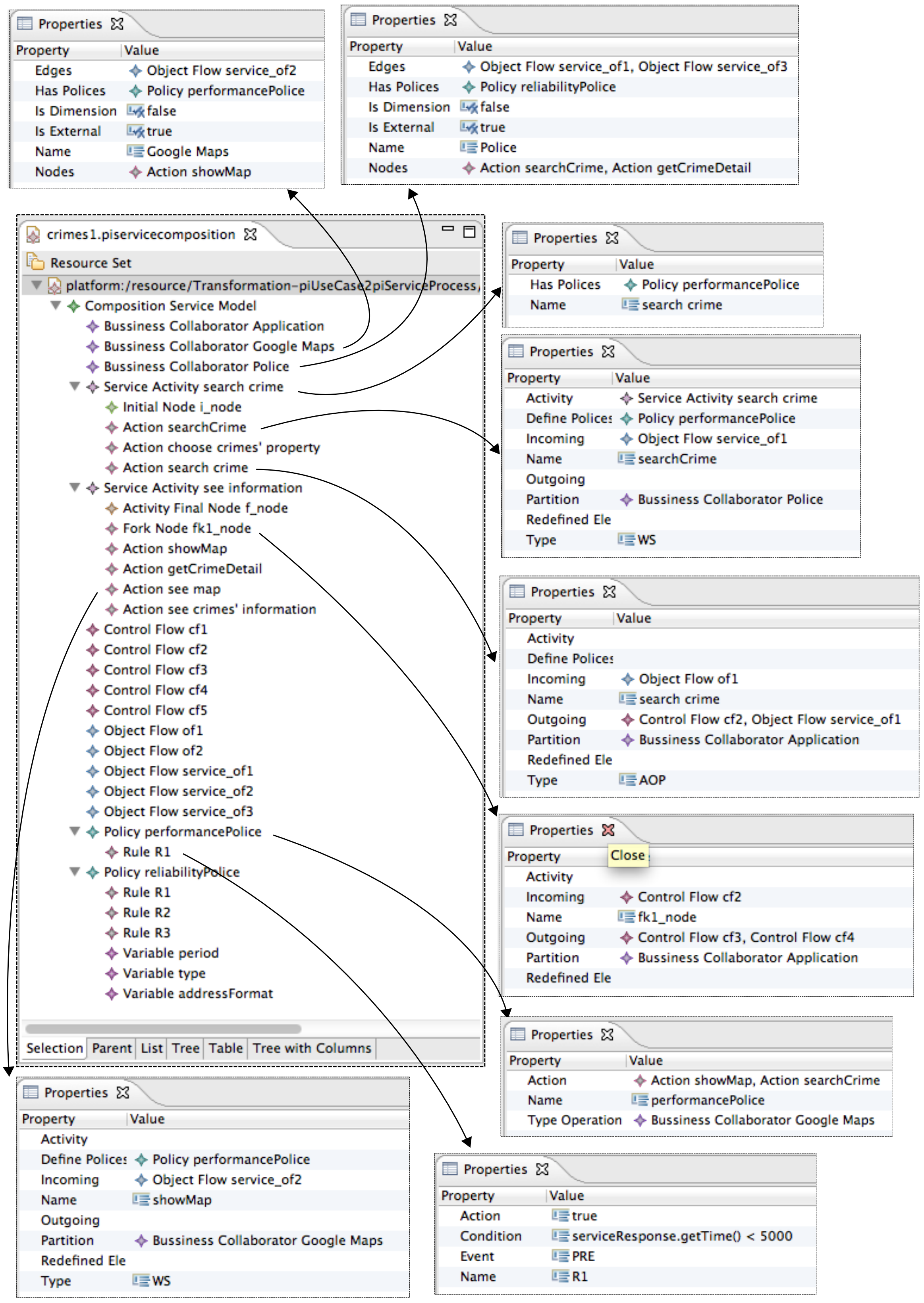
```
<<policy>>
$reliabilityPolicy$
%Reliability%

Rule R1{
@PRE@

#addressFormat == ???#

! abort()!
}
Rule R2{
@POST@

#not(getCrimes(types).length > 5)
--> false#

!show(getCrimesInformation())!
}
Rule R3{
@POST@

#result[i].period < 1800#

!show(getCrimesInformation())!
}
```

Figure 70: *Crime information* $\pi$-ServiceComposition.

all policies are applied to the entire actions associated with a service activity, for example, the *performancePolicy* verifies the request time for presenting the Google maps. If the time exceeds 5000 milliseconds, the maps call is ignored and only the crime information is presented. The *reliabilityPolicy* is associated with crime data verification. There are three rules over crime address format, types of crime and the period of time in which it happens. The crime presentation may obey the pre- and post-condition described in this policy. Figure 71 presents the $\pi$SOD-M tool description for the model described in Figure 70.

Figure 72 refines the $\pi$-ServiceProcess model described in Figure 68 and Figure 73 presents the $\pi$SOD-M tool description for the model described in Figure 72. This model details the policies for *crime statistic* and *share information* services. The policies are *statisticPolicy* and *authenticationPolicy*, respectively.

These models provide an overview of the external services and its composition from the actions described in the $\pi$-ServiceProcess model. These models also provide a more detailed



Properties

| Property | Value |
|---|---|
| Edges | Object Flow service_of2 |
| Has Polices | Policy performancePolice |
| Is Dimension | false |
| Is External | true |
| Name | Google Maps |
| Nodes | Action showMap |

Properties

| Property | Value |
|---|---|
| Edges | Object Flow service_of1, Object Flow service_of3 |
| Has Polices | Policy reliabilityPolice |
| Is Dimension | false |
| Is External | true |
| Name | Police |
| Nodes | Action searchCrime, Action getCrimeDetail |

crimes1.piservicecomposition

```
Resource Set
  platform:/resource/Transformation-piUseCase2piServiceProcess/
    Composition Service Model
      Bussiness Collaborator Application
      Bussiness Collaborator Google Maps
      Bussiness Collaborator Police
      Service Activity search crime
        Initial Node i_node
        Action searchCrime
        Action choose crimes' property
        Action search crime
      Service Activity see information
        Activity Final Node f_node
        Fork Node fk1_node
        Action showMap
        Action getCrimeDetail
        Action see map
        Action see crimes' information
      Control Flow cf1
      Control Flow cf2
      Control Flow cf3
      Control Flow cf4
      Control Flow cf5
      Object Flow of1
      Object Flow of2
      Object Flow service_of1
      Object Flow service_of2
      Object Flow service_of3
      Policy performancePolice
        Rule R1
      Policy reliabilityPolice
        Rule R1
        Rule R2
        Rule R3
        Variable period
        Variable type
        Variable addressFormat
```

Selection | Parent | List | Tree | Table | Tree with Columns

Properties

| Property | Value |
|---|---|
| Has Polices | Policy performancePolice |
| Name | search crime |

Properties

| Property | Value |
|---|---|
| Activity | Service Activity search crime |
| Define Police: | Policy performancePolice |
| Incoming | Object Flow service_of1 |
| Name | searchCrime |
| Outgoing | |
| Partition | Bussiness Collaborator Police |
| Redefined Ele | |
| Type | WS |

Properties

| Property | Value |
|---|---|
| Activity | |
| Define Police: | |
| Incoming | Object Flow of1 |
| Name | search crime |
| Outgoing | Control Flow cf2, Object Flow service_of1 |
| Partition | Bussiness Collaborator Application |
| Redefined Ele | |
| Type | AOP |

Properties

| Property | Close |
|---|---|
| Activity | |
| Incoming | Control Flow cf2 |
| Name | fk1_node |
| Outgoing | Control Flow cf3, Control Flow cf4 |
| Partition | Bussiness Collaborator Application |
| Redefined Ele | |

Properties

| Property | Value |
|---|---|
| Action | Action showMap, Action searchCrime |
| Name | performancePolice |
| Type Operation | Bussiness Collaborator Google Maps |

Properties

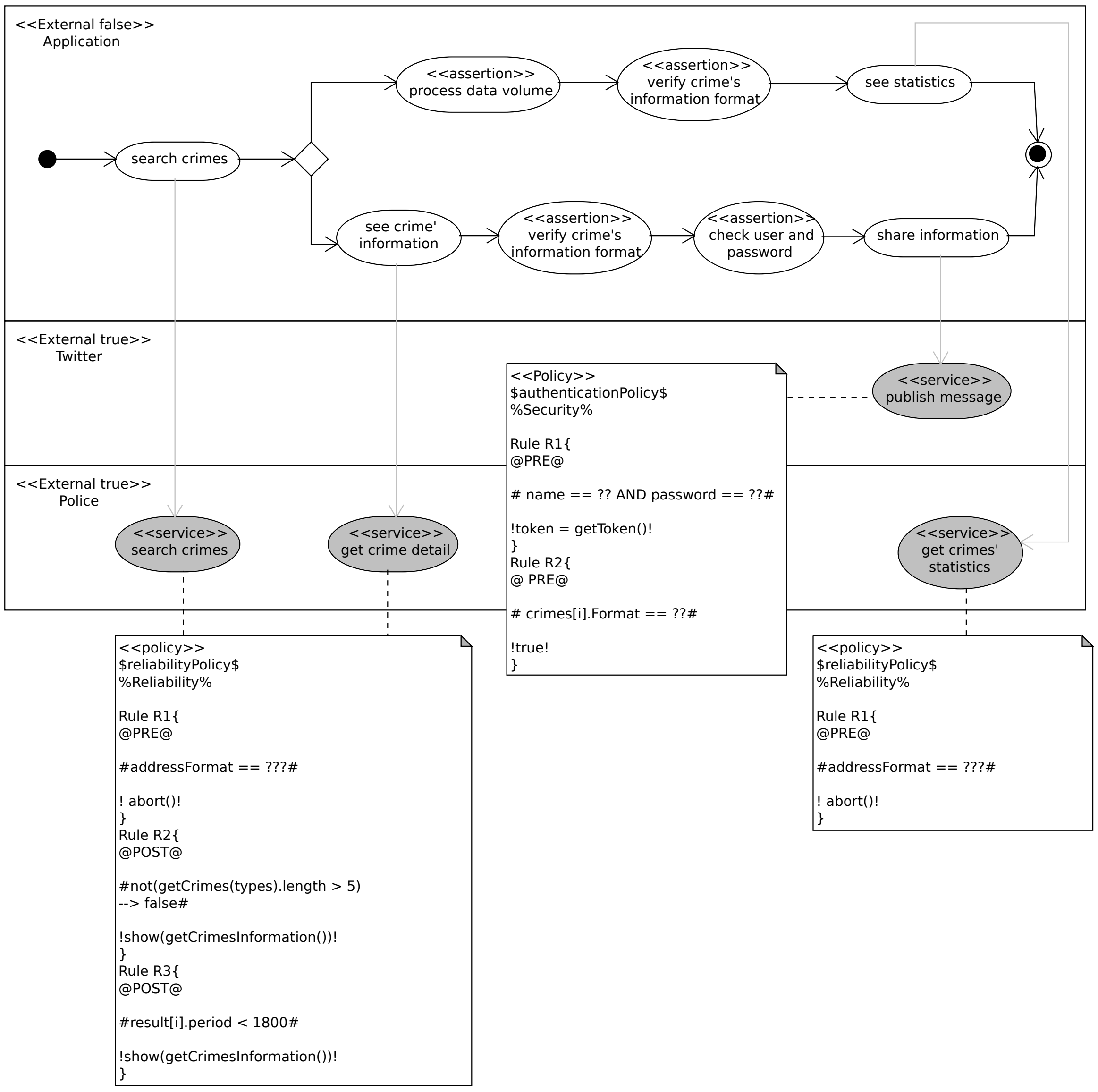
| Property | Value |
|---|---|
| Activity | |
| Define Police: | Policy performancePolice |
| Incoming | Object Flow service_of2 |
| Name | showMap |
| Outgoing | |
| Partition | Bussiness Collaborator Google Maps |
| Redefined Ele | |
| Type | WS |

Properties

| Property | Value |
|---|---|
| Action | true |
| Condition | serviceResponse.getTime() < 5000 |
| Event | PRE |
| Name | R1 |

Figure 71: *Crime information* $\pi$-ServiceComposition Environment.

view of the system restrictions, applying policies over external services. This model refines the $\pi$-ServiceProcess models detailing the Business Collaborators that are expressed as external



```
<<Policy>>
$authenticationPolicy$
%Security%

Rule R1{
@PRE@

# name == ?? AND password == ??#

!token = getToken()!
}
Rule R2{
@ PRE@

# crimes[i].Format == ??#

!true!
}
```

```
<<policy>>
$reliabilityPolicy$
%Reliability%

Rule R1{
@PRE@

#addressFormat == ???#

! abort()!
}
Rule R2{
@POST@

#not(getCrimes(types).length > 5)
--> false#

!show(getCrimesInformation())!
}
Rule R3{
@POST@

#result[i].period < 1800#

!show(getCrimesInformation())!
}
```

```
<<policy>>
$reliabilityPolicy$
%Reliability%

Rule R1{
@PRE@

#addressFormat == ???#

! abort()!
}
```

Figure 72: *Crime statistic and share information* $\pi$-ServiceComposition.

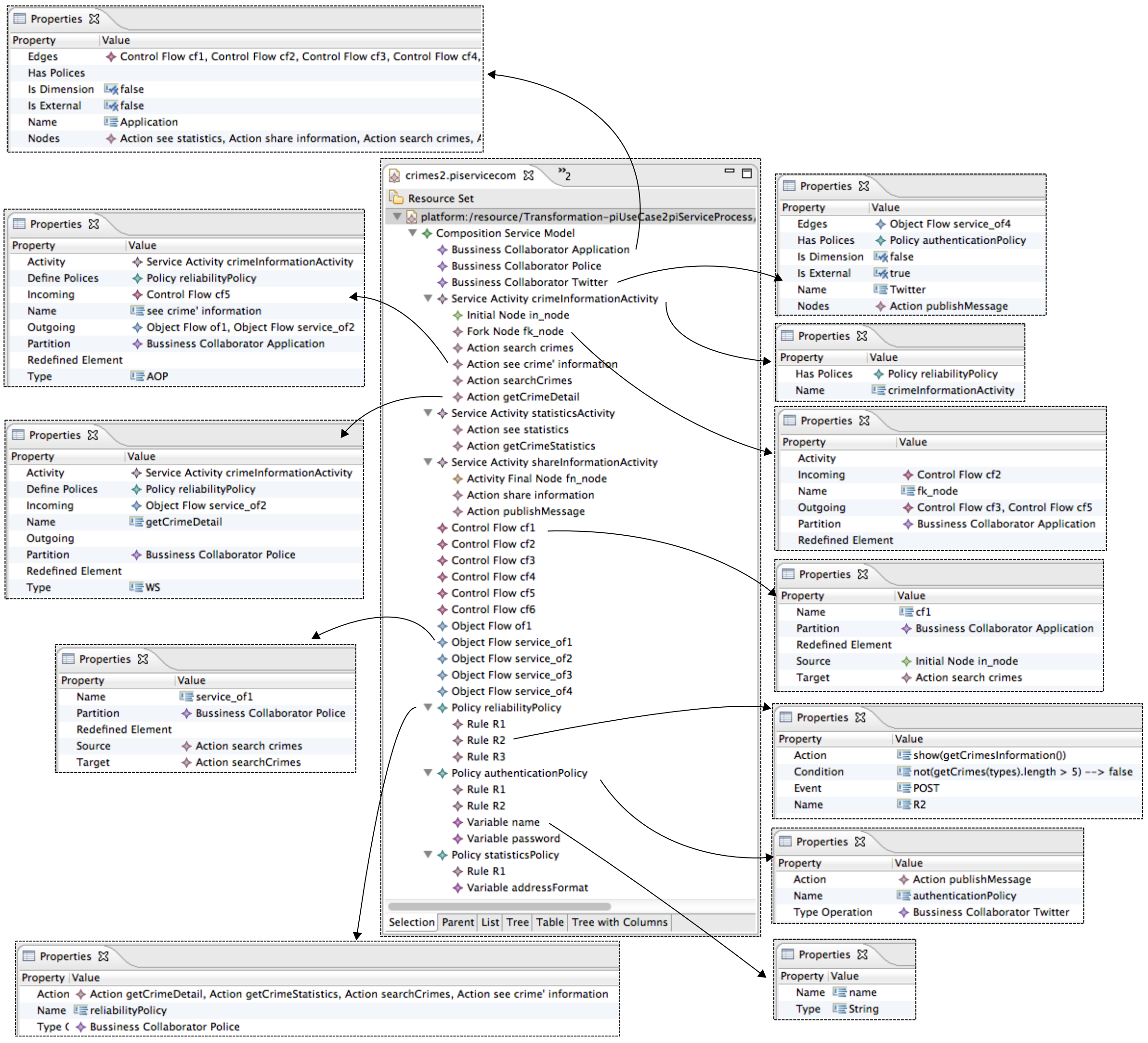
services, and grouping contract into policies.

The result of the transformation of this model to the $\pi$-*PEWS* model is gives in the listing 5.1 and 5.2.

## 5.3 Case Study 3: GesIMED Application

Aiming to perform a qualitative analysis and the validation of $\pi$SOD-M methodology, we present the *GesIMED Application*[2] case study.

[2] If necessary, a detailed description of this case study can be found in [36].



```
crimes2.piservicecom
Resource Set
▼ platform:/resource/Transformation-piUseCase2piServiceProcess
  ▼ Composition Service Model
      Bussiness Collaborator Application
      Bussiness Collaborator Police
      Bussiness Collaborator Twitter
    ▼ Service Activity crimeInformationActivity
        Initial Node in_node
        Fork Node fk_node
        Action search crimes
        Action see crime' information
        Action searchCrimes
        Action getCrimeDetail
    ▼ Service Activity statisticsActivity
        Action see statistics
        Action getCrimeStatistics
    ▼ Service Activity shareInformationActivity
        Activity Final Node fn_node
        Action share information
        Action publishMessage
      Control Flow cf1
      Control Flow cf2
      Control Flow cf3
      Control Flow cf4
      Control Flow cf5
      Control Flow cf6
      Object Flow of1
      Object Flow service_of1
      Object Flow service_of2
      Object Flow service_of3
      Object Flow service_of4
    ▼ Policy reliabilityPolicy
        Rule R1
        Rule R2
        Rule R3
    ▼ Policy authenticationPolicy
        Rule R1
        Rule R2
        Variable name
        Variable password
    ▼ Policy statisticsPolicy
        Rule R1
        Variable addressFormat
```

| Property | Value |
|---|---|
| Edges | Control Flow cf1, Control Flow cf2, Control Flow cf3, Control Flow cf4, |
| Has Polices | |
| Is Dimension | false |
| Is External | false |
| Name | Application |
| Nodes | Action see statistics, Action share information, Action search crimes, A |

| Property | Value |
|---|---|
| Activity | Service Activity crimeInformationActivity |
| Define Polices | Policy reliabilityPolicy |
| Incoming | Control Flow cf5 |
| Name | see crime' information |
| Outgoing | Object Flow of1, Object Flow service_of2 |
| Partition | Bussiness Collaborator Application |
| Redefined Element | |
| Type | AOP |

| Property | Value |
|---|---|
| Activity | Service Activity crimeInformationActivity |
| Define Polices | Policy reliabilityPolicy |
| Incoming | Object Flow service_of2 |
| Name | getCrimeDetail |
| Outgoing | |
| Partition | Bussiness Collaborator Police |
| Redefined Element | |
| Type | WS |

| Property | Value |
|---|---|
| Name | service_of1 |
| Partition | Bussiness Collaborator Police |
| Redefined Element | |
| Source | Action search crimes |
| Target | Action searchCrimes |

| Property | Value |
|---|---|
| Action | Action getCrimeDetail, Action getCrimeStatistics, Action searchCrimes, Action see crime' information |
| Name | reliabilityPolicy |
| Type ( | Bussiness Collaborator Police |

| Property | Value |
|---|---|
| Edges | Object Flow service_of4 |
| Has Polices | Policy authenticationPolicy |
| Is Dimension | false |
| Is External | true |
| Name | Twitter |
| Nodes | Action publishMessage |

| Property | Value |
|---|---|
| Has Polices | Policy reliabilityPolicy |
| Name | crimeInformationActivity |

| Property | Value |
|---|---|
| Activity | |
| Incoming | Control Flow cf2 |
| Name | fk_node |
| Outgoing | Control Flow cf3, Control Flow cf5 |
| Partition | Bussiness Collaborator Application |
| Redefined Element | |

| Property | Value |
|---|---|
| Name | cf1 |
| Partition | Bussiness Collaborator Application |
| Redefined Element | |
| Source | Initial Node in_node |
| Target | Action search crimes |

| Property | Value |
|---|---|
| Action | show(getCrimesInformation()) |
| Condition | not(getCrimes(types).length > 5) --> false |
| Event | POST |
| Name | R2 |

| Property | Value |
|---|---|
| Action | Action publishMessage |
| Name | authenticationPolicy |
| Type Operation | Bussiness Collaborator Twitter |

| Property | Value |
|---|---|
| Name | name |
| Type | String |

Figure 73: *See crime statistic and share information* π-ServiceComposition Environment Detail.

This case study was originally developed in [36] for the SOD-M methodology. Here , we adapt the requirements of the system to include NFR. This system is a management Web system that processes medical images through the Web [36]. The objective is to manage information about the creation and maintenance of scientific studies for neuroscience research. The application is designed to be used primarily by researchers in neuroscience, such as neurologists, neuropsychologists and neuroradiologists who conducting research in this area.

The application requirements are the following: (i) a database of medical images that can be accessed by the user; (ii) it has an interface for querying the database; (iii) implement standard procedures for analyzing and processing stored images; and (iv) the images analysis and processing results must be also stored, so they can be used in future studies.

The Medical Image Analysis Laboratory from the Universidad Rey Juan Carlos (LAIM)

Listing 5.1: pi-PEWS Specification: Crime information

```
ns police =  "http:\\www.pm.rn.gov.br/service"
ns googleMaps =  "http://maps.googleapis.com/maps/api/"

alias chooseCrimesProperties = portType/chooseCrimesProperties in police
alias searchCrimes = portType/searchCrimes in police
alias seeMap = portType/showMap in googleMaps
alias seeCrimesInformation = portType/getCrimeDetail in police

(chooseCrimesProperties . searchCrimes)* . seeMap . seeCrimesInformation

def contract searchCrimesContract{
 isAppliedTo: searchCrimes;
 requires: addressFormat == ??
    (onFailureDo: call(chooseCrimesProperties) );
   ensures : !(getCrimes(types).length > 5) ==> false &&
              result[i].period < 1800
      (onFailureDo: show(getCrimesInformation()));
}

def contract showMapContract{
 isAppliedTo: seeMap;
   ensures : serviceResponse.getTime() < 5000
      (onFailureDo: skip);
}
```

Listing 5.2: pi-PEWS Specification: Crime statistic and share information

```
ns police =  "http:\\www.pm.rn.gov.br/service"
ns twitter =  "https://dev.twitter.com/docs/api/"

alias searchCrimes = portType/searchCrimes in police
alias seeCrimesInformation = portType/getCrimeDetail in police
alias seeStatistics = portType/getCrimeStatistics in police
alias shareInformation = portType/publishMessage in twitter

searchCrimes . ((seeCrimesInformation . shareInformation) |
                                        seeStatistics)

def contract searchCrimesContract{
 isAppliedTo: searchCrimes;
 requires: addressFormat == ??
    (onFailureDo: call(chooseCrimesProperties) );
   ensures : !(getCrimes(types).length > 5) ==> false &&
              result[i].period < 1800
      (onFailureDo: show(getCrimesInformation()));
}

def contract shareInformationContract{
 isAppliedTo: shareInformation;
 requires: name == ?? AND password == ?? &&
              crimes[i].Format == ??
    (onFailureDo: call(shareInformation) );
}

def contract seeStatisticsContract{
 isAppliedTo: seeStatistics;
 requires: crimes[i].Format == ??
    (onFailureDo: call(seeStatistics) );
}
```

offers three specific services to researchers in neuroscience. The access to services has a cost and financial revenues are assigned to LAIM. The services are:

- Storage and retrieval medical imaging service (SACim);
- Image processing service (SPim);
- Image visualization service (SVim);

From the functions offered by these services we will detail the application characteristics. The application uses these services offered by LAIM for modeling the application.

Given the description of the business requirements and the expected functionality of the system, $\pi$SOD-M is used for developing this application.

We will detail the *perform image processing* business service. In $\pi$SOD-M, the $\pi$-*UseCase* model describes the application functions, restrictions and its non-functional attributes. This business service (*perform image processing*) is sufficiently rich to describe the peculiarities of $\pi$SOD-M compared to the original model of SOD-M.

## 5.3.1 $\pi$-*UseCase* Model

Figure 74 shows the $\pi$-*UseCase* model. This model defines four constraints: *authenticate, response time, data format* and *validate payment card*. These constraints are restrictions over the functions of the *GesIMED* application. In the description of each constraint it is necessary to describe the constraint type (*business* or *value*). The model has one business constraint and three value constraints. It is possible to describe constraints for each use case application.

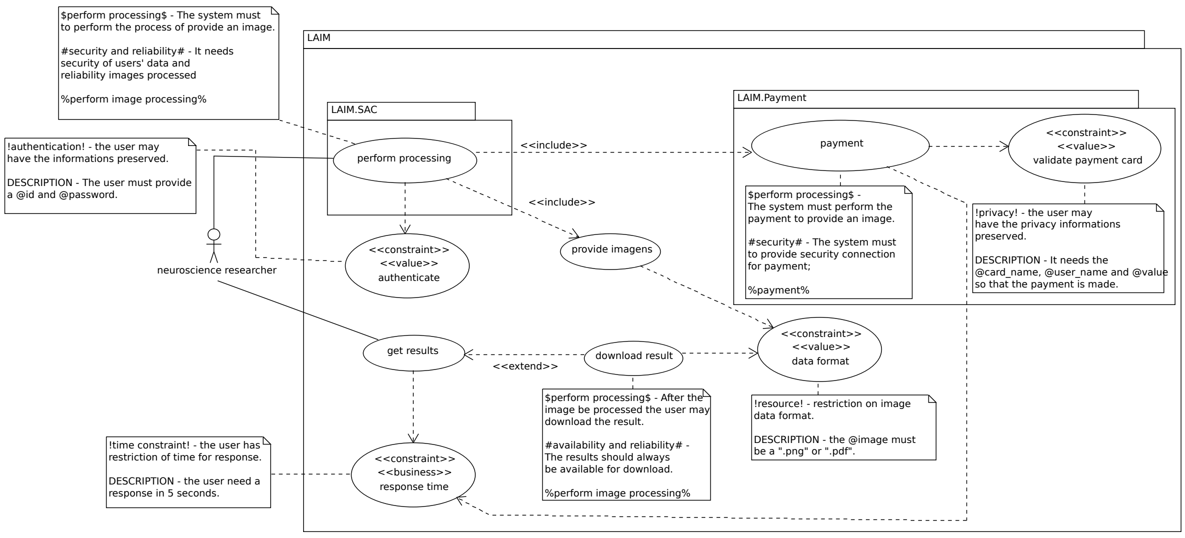

$perform processing$ - The system must to perform the process of provide an image.
#security and reliability# - It needs security of users' data and reliability images processed
%perform image processing%
LAIM
LAIM.SAC
LAIM.Payment
!authentication! - the user may have the informations preserved.
DESCRIPTION - The user must provide a @id and @password.
perform processing
<<include>>
payment
<<constraint>> <<value>> validate payment card
<<include>>
$perform processing$ - The system must perform the payment to provide an image.
#security# - The system must to provide security connection for payment;
%payment%
!privacy! - the user may have the privacy informations preserved.
DESCRIPTION - It needs the @card_name, @user_name and @value so that the payment is made.
neuroscience researcher
<<constraint>> <<value>> authenticate
provide imagens
<<constraint>> <<value>> data format
get results
<<extend>>
download result
$perform processing$ - After the image be processed the user may download the result.
#availability and reliability# - The results should always be available for download.
%perform image processing%
!resource! - restriction on image data format.
DESCRIPTION - the @image must be a ".png" or ".pdf".
!time constraint! - the user has restriction of time for response.
DESCRIPTION - the user need a response in 5 seconds.
<<constraint>> <<business>> response time

Figure 74: $\pi$-UseCase: Perform Image Processing.

Besides the constraints, the model details the non-functional attributes related to each constraint, for example, the *response time* constraint. For value constraints, it is necessary identify candidates variables to be verified, through the use of @. For example, the authentication constraint, has the @*id* and @*password* identifiers that must be verified.

The main feature of our approach is the association of non-functional requirements with use cases. For example, the user can identify that the quality requirement related to a payment process is *transactional* or *data privacy*. So, through the process, it is possible to refine this information, considering other details provided.

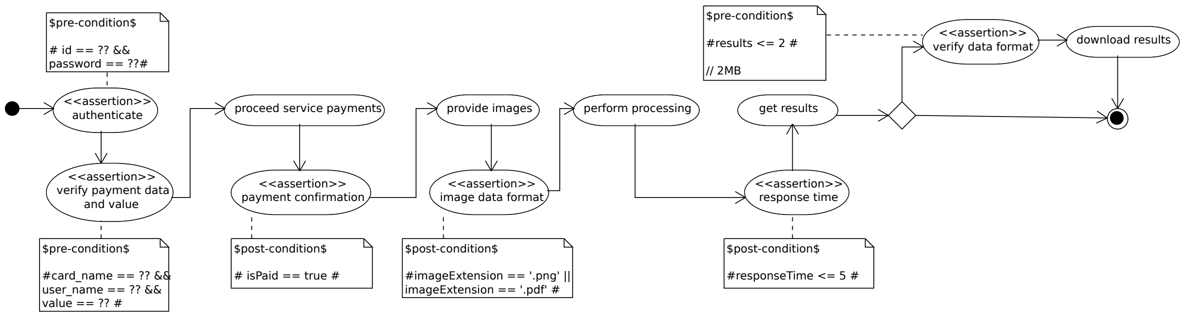


Figure 75: $\pi$-ServiceProcess: Perform Image Processing.

The result of the $\pi$-*UseCase* model is a detailed list of constraints, its respective use case, system requirement information and non-functional attributes required for each service.

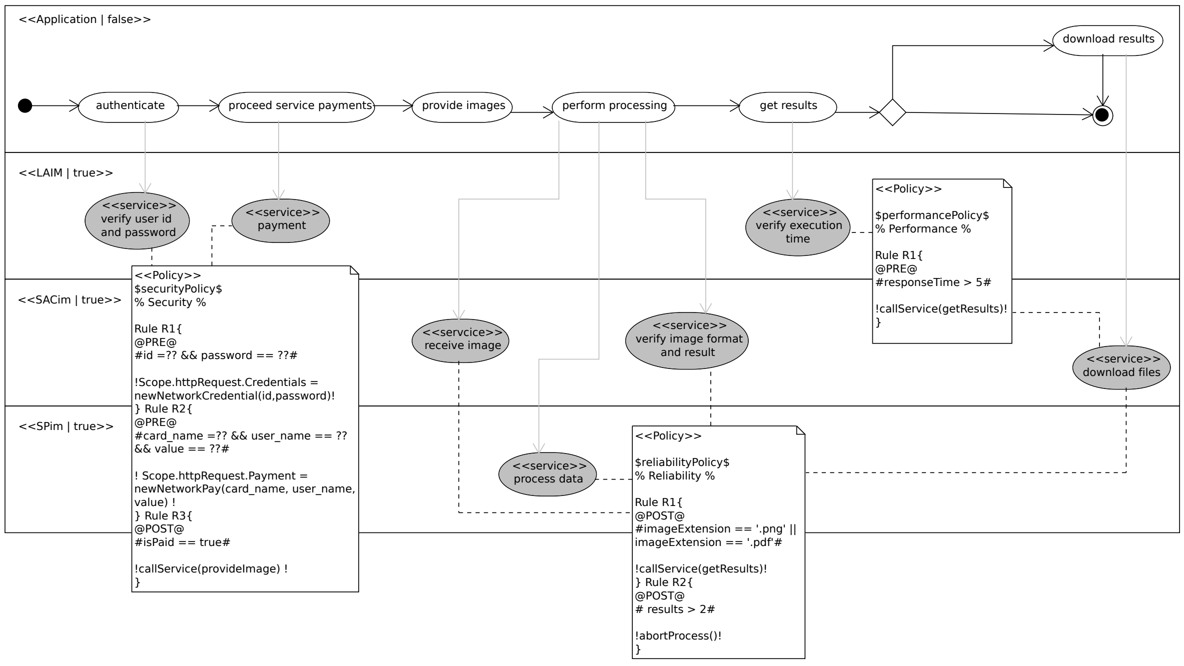
### 5.3.2 $\pi$-*ServiceProcess* Model

Figure 75 shows the $\pi$-*ServiceProcess* model. This model transforms the use cases into actions, and constraints into assertions. The resulting workflow describes pre or post-conditions on workflow execution. The assertions are wrappers around the action. All the assertions related to a specific action become a contract. For example, the *proceed payments service* action has a pre- and a post-condition. The pre-condition is a restriction over the payment information (*card number, name* and *value*), while the post-condition defines the final state after the payment. The other assertions of this model are the result of the transformation of each constraint described in the $\pi$-*UseCase* model.

### 5.3.3 $\pi$-*ServiceComposition* Model

The $\pi$-*ServiceComposition* model presented in figure 76 refines the $\pi$-*ServiceProcess* model. This model describes the real services and their associated collaborators. The information of the business collaborator comes from the package description. Contracts with the same non-functional requirements become service policies. The policies define rules, which are a direct transformation of the assertions in the previous model.

Listing 5.3 shows the $\pi$-*PEWS* specification that was generated from the $\pi$-*ServiceComposition* model. This specification considers the operations (lines 6-16), its business collaborators (namespaces in the lines 1-4), the main workflow (lines 18-20) and services restrictions (described as contracts in the lines 21-40). The final result of our approach is the specification that describes the structure and sequence of execution to be performed by an orchestrator. $\pi$-*PEWS* specification defines a set of contracts to represent the application restric-



```
<<Policy>>
$securityPolicy$
% Security %

Rule R1{
@PRE@
#id =?? && password == ??#

!Scope.httpRequest.Credentials =
newNetworkCredential(id,password)!
} Rule R2{
@PRE@
#card_name =?? && user_name == ??
&& value == ??#

! Scope.httpRequest.Payment =
newNetworkPay(card_name, user_name,
value) !
} Rule R3{
@POST@
#isPaid == true#

!callService(provideImage) !
}
```

```
<<Policy>>

$performancePolicy$
% Performance %

Rule R1{
@PRE@
#responseTime > 5#

!callService(getResults)!
}
```

```
<<Policy>>

$reliabilityPolicy$
% Reliability %

Rule R1{
@POST@
#imageExtension == '.png' ||
imageExtension == '.pdf'#

!callService(getResults)!
} Rule R2{
@POST@
# results > 2#

!abortProcess()!
}
```

Figure 76: $\pi$-ServiceComposition: Perform Image Processing.

tions. Thus, using $\pi$SOD-M it is possible to refine the models until the code generation.

## 5.4 Lessons Learned

This section summarizes the experience we obtained after the use of $\pi$SOD-M for the three case studies.

### 5.4.1 Case Study 1: To Publish Music

In this case study we can see that the restrictions on web services can be modeled from the early stages of development, not leaving it to the programming language, or to the programmer to resolve this type of problem. More specifically, this case study had restrictions *performance* and *security*. Since the $\pi$-*UseCase* model to the $\pi$-*ServiceComposition* model, non-functional requirements and their attributes might be described without serious impact on the modeling of other application requirements.

The restrictions on the functions provided by the *Spotify* service was basically *performance*. This type of restriction is not easy to represent in the implementation, because it relies on the network available to the user. We described the constraints and refined them into a *performance* policy for updating the music. The security restrictions have been described for the service that provides banking information. The security policy generated was described for two functions of

Listing 5.3: pi-PEWS Specification: Perform Image Processing

```
ns app =  "http:\\www.neuro.laim.org"
ns LAIM =  "http:\\www.neuro.laim.org/laim—service.wsdl"
ns SACim = "http:\\www.neuro.laim.org/sacim—service.wsdl"
ns SPim =  "http:\\www.neuro.laim.org/spim—service.wsdl"

alias authenticate = portType/verifyUserandPassword in LAIM
alias proceedServicePayment = portType/payment in LAIM
alias provideImage = portType/provideImage in app
alias receiveImage = portType/receiveImage in SACim
alias processData = portType/processData in SPim
alias verifyImageFormatAndResult = portType/verifyImageFormatAndResult in
SACim alias getResults = portType/verifyExecutionTipe in LAIM
alias downloadResults = portType/downloadFiles in SACim

service performProcessing =
            receiveImage . processData . verifyImageFormatAndResult

authenticate . proceedServicePayment . provideImage .
  performProcessing . [act(getResults).time > 5]getResults .
  {downloadResults}

def contract authenticateContract{
 isAppliedTo: authenticate;
 requires: id == ?? && password = ??;
    (onFailureDo: call(authenticate) )
}

def contract paymentContract{
 isAppliedTo: proceedServicePayment;
 requires: card_name =?? &&
                      user_name == ?? &&
                      value == ??;
 ensures : isPaid == true ;
}

def contract performProcessingContract{
 isAppliedTo: performProcessing;
 requires: imageExtension == '.png' || imageExtension == '.pdf'
   (onFailureDo: skip );
 ensures : resultValue <= 2 ;
}
```

the Bank service, *pay* and *send confirmation* functions. This type of specification can be refined incrementally by applying the modelling steps defined by the $\pi$SOD-M methodology.

This case study does not cover all the possible restrictions in an application but it shows that it is possible to guarantee quality requirements refining information at each iteration of the modeling, such as constraints of *performance* and *security*.

The difficulty in modeling this case study was the definition of policies for each type of non-functional requirement. For example, the *httpAuthPolicy* policy for the authentication on Facebook service has restrictions from both types, *performance* and *security*.

### 5.4.2 Case Study 2: Crime Map

The most important process when designing a system is the requirement analysis process, especially identifying the non-functional requirements and the system restrictions. In order to ease the coding process and let the developer focus on technical issues, the system architecture must be automatically generated, from the system design.

Using $\pi$SOD-M has enabled the explicit specification of functional and non-functional requirements and their refinement along the different phases of the development process.

In this case study we can see the restrictions on four web services: *Police, Google Maps, Twitter* and *Post* service. As these services are independent, the constraints of the application being developed must comply with the service APIs. If a restriction is inconsistent with a function, there is no guarantee that it will be respected. For example, if the police offers crimes information on a particular region by type or day, it is not possible to present the restrictions of a particular time of day, for example, crimes that happen in the afternoon.

This case study has two *reliability* policies and one *security* policy. Policies on data are common, because they are restrictions on the service interface. We notice in the development of this case study that function restrictions, especially the execution time and time response are more difficult to ensure.

### 5.4.3 Case Study 3: GesIMED

The $\pi$SOD-M methodology helps to define a detailed description of the application, its restrictions and non-functional requirements associated with each service. The refinement of the quality requirements of each level helps to improve the development detail in order to produce a more reliable application. The NFRs are not only described in a general way, but the restrictions are designed to each service individually.

Considering the modeling of this case study in SOD-M, the first step of the is to list the global requirements (business case). Figure 77 shows the use case model for the application. This model is very simple and describes the main application requirements. The actor is represented as the “neuroscience researcher”, which is the final consumer of the services to be implemented. Furthermore, the model is represented as use cases, which describes the main system business services: “perform image processing[3]”, “perform image view” and “perform query”.

The next model is the the extended use cases model. This model is used to model the functionality required by the system to perform each business services. Business services described in the previous model, become more detailed in the extended use case model. In this model are described both the application’s features such as the possible services to be performed. Figure 78 presents the extended use case model for *perform image processing* business service. This model is detailed in 6 use cases that are necessary for the image processing. The use cases are:

[3]During our analysis, we will detail the *perform image processing* requirement, as a way to better understand the particularities of both methods.

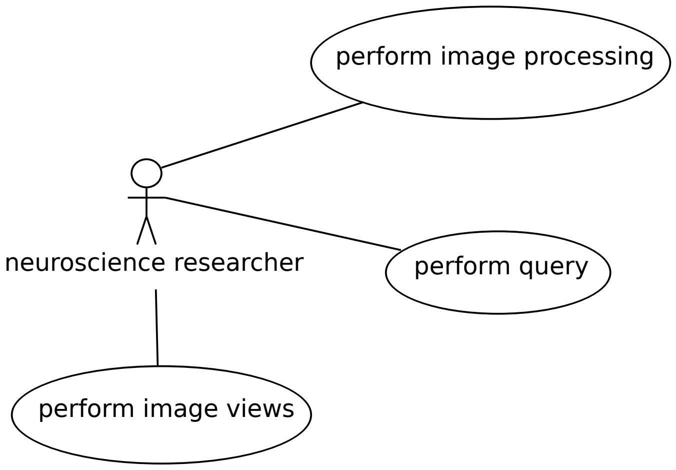


Figure 77: Use Case Model [36].

(i) perform image processing; (ii) proceed with the service payment; (iii) provide images; (iv) authenticate; (v) get results; and (vi) download results.

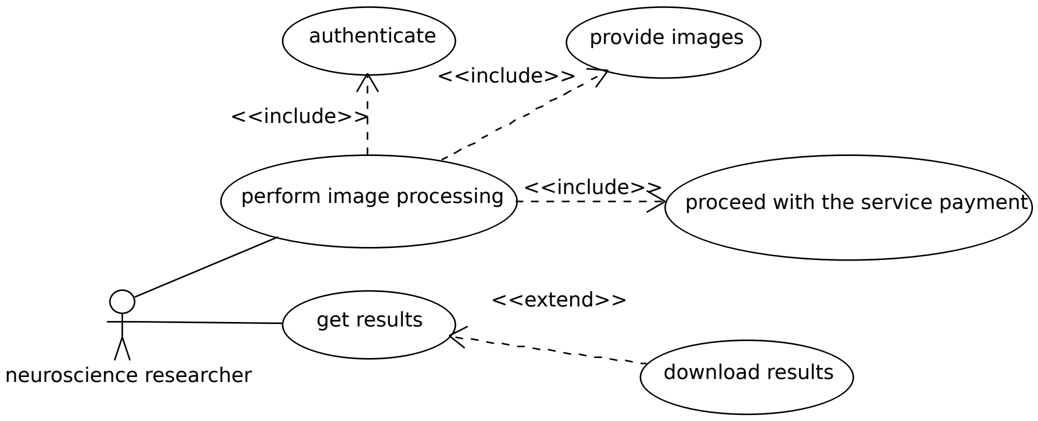


Figure 78: Extended Use Case Model [36].

Once identified the different functionalities that are required by the system to perform business services, the service process model is used to represent the workflow necessary to perform a business service. SOD-M represents each business service in different diagrams, which are shown in the following figures. Figure 79, for the business service "*perform image processing*". The activities represented in these figures are derived from the basic use cases represented in the model extended use case.

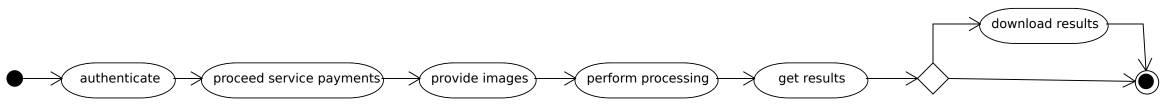


Figure 79: Image Processing - Service Process Diagram [36].

The service composition models represent the processes of composition of the various actions that are necessary to perform each business service, detailing which business collaborator performing each action.

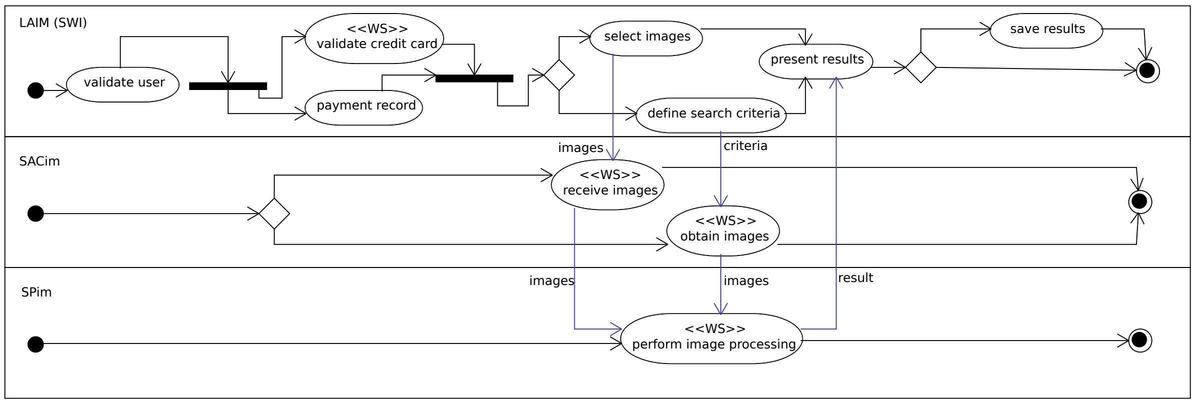


Figure 80: Image Processing - Service Composition Diagram [36].

Figure 80 present the details of the services composition for the business service we are modelling. It is important to realize that the service processes described previously, are refined by the execution of these service. Notice that SOD-M creates a workflow for each business service, independently. This workflow is a sequence of services that can be run considering the system features. Figure 80 presents that the workflow uses the services offered by LAIM to perform each function, according the SACim, SPim and SVim services. It is also important to note that the external application services are marked with the stereotype *«WS»*.

The result of applying the SOD-M method is a detailed service composition model from the business use case, in our specific case the "*perform image processing*" business service.

Notice that the non-functional properties is not modelled by SOD-M, in contract to $\pi$SOD-M models.

## 5.5 Conclusions

This chapter presented the use of the $\pi$SOD-M methodology, as a form of empirical validation. We described, modeled and implemented three case studies to use concepts, model properties and detail the process development. The methodology and its concepts supported the development of applications that use service compositions, mainly in the identification and refinement of quality requirements. The use of $\pi$SOD-M and its environment can help to identify the non-functional requirements and improve the system modeling and specification process.

The transformations of models enables the semi-automatic development of applications. Owing to the generation of the system specification code from the service composition model, the designer and developer can focus on the description and detail of functional and non-functional requirements. The generated code contains the expression of the interaction with services and their associated namespaces. This specification can be executed after the model transformation.

As the process is iterative and incremental, after the generation of each model it is possible to make adjustments to the generated model. For example, after the $\pi$-UseCase model design and the generation of $\pi$-ServiceProcess model, the designer can make adjustments and improve the model so that it becomes more expressive.

# 6 Conclusions

Today, access to software, hardware and network resources is done more and more through services. Building applications implies composing these services according to given requirements. Applications must ensure a certain level of compliance to specified requirements and reliability properties to ensure their quality. Ensuring compliance and reliability of applications composed by services provided under different conditions and with different functions and properties is not an easy task and imposes challenges for systems developers.

Non-functional requirements are related to business rules associated to the general semantics of the application and in the case of service based applications, they also concern the use constraints imposed by the services. Having such business rules expressed and then translated and associated to the service composition can help to ensure that the resulting application fulfills user requirements and it considers the characteristics of the services it uses.

The main goal of this work is to provide tools for easing the development of reliable service-based applications. Our work enables the specification and programming of non-functional aspects (i.e., atomicity, security, exception handling, persistence). In contrast to approaches such as WS-*, our work specifies policies for a service composition in an orthogonal way. Besides, these approaches suppose that non-functional requirements are implemented according a the knowledge that a programmer has of a specific application requirements but they are not derived in a methodological way, leading to ad-hoc solutions that can be difficult to reuse. In our approach, once defined policies for a given application they can be reused and/or specialized for another one with the same requirements or that uses services that impose the same constraints.

## 6.1 Main Contributions

This thesis presented a methodology for specifying and designing reliable service based applications. We model and associate policies to service-based applications that represent both systems' cross-cutting aspects and use constraints stemming from the services used for implementing them.

We provide a platform for implementing the methodology, implemented through an Eclipse plugin environment.

$\pi$SOD-M is proposed for the developing of service-oriented applications, helping to ensure quality requirements. This work also proposes a set of concepts for the modeling and refinement of non-functional properties, such as, *constraint, contract, assertion, policy, rules, non-functional requirement* and *non-functional attribute*.

Evaluations were made, showing that the development process based on $\pi$SOD-M produces satisfactory results with respect to the modeling and refinement of quality requirements. We also noticed that grouping contracts with non-functional attributes into a single policy, produces a more effective result in the generation of the specification, and therefore in the verification of these properties.

Another contribution of our proposal is the integration of the methodology concepts in a MDA-based development. $\pi$SOD-M is a method that proposes meta-models at different levels (CIM, PIM and PSM) and extending the PSM meta-models. It enables the design and development of service-based applications that can be reused.

We also extended the PEWS language [80] (into $\pi$-PEWS) to add the contract specification and temporal modeling restrictions for services. We also extended the plugin for specification and generation of XML representation of PEWS [27]. The plugin also supports the verification of each service functions [39], whether it is active or not. This tool supports the methodology after the generation of the specification from the models described.

## 6.2 Future Work

This thesis proposes a comprehensive methodology that can offer a range of possibilities for extension in future work. Future work is organized into *validation*, *development* and *research tasks*:

- Validation:
  - Validate $\pi$SOD-M to perform a more quantitative analysis;
  - Compare the generated code and implementation specifications with different languages;

  - Describe software product lines related with reliable services applications that could be generated from the use of $\pi$SOD-M;
  - Analyze the variability of non-functional requirements proposed by $\pi$SOD-M.

- Development:
  - Develop the $\pi$-PEWS environment (back-end) and then run validation experiments again;
  - Improve development environment that support the methodology, with visual tools and with execution environment, running in different platforms.

- Research tasks:
  - Define meta-models at the PSM level for generating code in other languages;
  - Formalize the transformation rules at different levels of the methodology;
  - Define of meta-models in the CIM level to represent business and system requirements, despite the original proposal offer *e-value* and *BPMN* models. It would be important to do a more thorough search in regarding these requirements to represent the independent level computing.

# *References*

[1] *The Rational Unified Process Made Easy: A Practitioner's Guide to the RUP: A Practitioner's Guide to the RUP*. Addison-Wesley, 2003.

[2] *Business Process Management, Second Edition: Practical Guidelines to Successful Implementations*. Elsevier, 2008.

[3] Arsanjani A., Ghosh S., Allam A., Abdollah T., Ganapathy S., and Holley K. SOMA: A method for developing service-oriented solutions. *IBM System Journal*, 47(3), 2008.

[4] Brown et. al A. SOA Development Using the IBM Rational Software Development Platform: A Practical Guide. In *Rational Software*, 2005.

[5] S. Abiteboul, O. Benjelloun, and T. Milo. Towards a flexible model for data and web services integration. In *proc. Internat. Workshop on Foundations of Models and Languages for Data and Objects*, 2001.

[6] Jean-Raymond Abrial, Matthew K. O. Lee, David Neilson, P. N. Scharbach, and Ib Holm Sï¿½rensen. The b-method. In Sï¿½ren Prehn and W. J. Toetenel, editors, *VDM Europe (2)*, volume 552 of *Lecture Notes in Computer Science*, pages 398–405. Springer, 1991.

[7] Sudhir Agarwal, Steffen Lamparter, and Rudi Studer. Making web services tradable: A policy-based approach for specifying preferences on web service properties. *J. Web Sem.*, 7(1):11–20, 2009.

[8] James F. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843, 1983.

[9] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Didier Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weeranwarana. Bussiness process execution language for web services. Available at http://www-128.ibm.com/developerworks/library/specification/ws-bpel/, 2003.

[10] Assaf Arkin, Sid Askary, Scott Fordin, Wolfgang Jekeli, Kohsuke Kawaguchi, David Orchard, Stefano Pogliani, Karsten Riemer, Susan Struble, Pal Takacsi-Nagy, Ivana Trickovic, and Sinisa Zimek. Web service choreography interface. Technical report, World Wide Web Consortium, 2002.

[11] Ali Arsanjani. SOMA: Service-Oriented Modeling and Architecture. Technical report, IBM, DisponÃvel em <http://www.ibm.com/developerworks/library/ws-soa-design1/>, 2004.

[12] Daniel Austin, Abbie Barbir, Ed Peters, and Steve Ross-Talbot. Web services choreography requirements. Available at http://www.w3.org/TR/2004/WD-ws-chor-reqs-20040311/, March 2004. W3C Working Draft.

[13] Cheikh Ba, Mirian Halfeld Ferrari, and Martin A. Musicante. Composing web services with PEWS: A trace-theoretical approach. In *IEEE European Conference on Web Services (ECOWS)*, pages 65–74, 2006.

[14] S.M. Babamir, S. Karimi, and M.R. Shishechi. A broker-based architecture for quality-driven web services composition. In *Computational Intelligence and Software Engineering (CiSE), 2010 International Conference on*, pages 1 –4, dec. 2010.

[15] Arindam Banerji, Claudio Bartolini, Dorothea Beringer, Venkatesh Chopella, Kannan Govindarajan, Alan Karp, Harumi Kuno, Mike Lemon, Gregory Pogossiants, Shamik Sharma, and Scott Williams. Web services conversation language (wscl) 1.0. Available at http://www.w3.org/TR/2002/NOTE-wscl10-20020314/, 2002.

[16] David A. Basin, Jürgen Doser, and Torsten Lodderstedt. Model driven security: From uml models to access control infrastructures. *ACM Trans. Softw. Eng. Methodol.*, 15(1):39–91, 2006.

[17] Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. Owl web ontology language. Technical report, W3C - http://www.w3.org/TR/owl-ref/, 2004.

[18] Khalid Belhajjame, Christine Collet, and Genoveva Vargas-Solar. A flexible workflow model for process-oriented applications. In *WISE (1)*, pages 72–, 2001.

[19] M. Bell. *Service-Oriented Modeling: Service Analysis, Design, and Architecture*. 2008.

[20] Michael Bell. *Service-Oriented Modeling (SOA): Service Analysis, Design, and Architecture*. John Wiley, 2008.

[21] Domenico Bianculli, Carlo Ghezzi, Paola Spoletini, Luciano Baresi, and Sam Guinea. A guided tour through savvy-ws: A methodology for specifying and validating web service compositions. In *Lipari Summer School*, pages 131–160, 2007.

[22] Egon Börger and Antonio Cisternino, editors. *Advances in Software Engineering, Lipari Summer School 2007, Lipari Island, Italy, July 8-21, 2007, Revised Tutorial Lectures*, volume 5316 of *Lecture Notes in Computer Science*. Springer, 2008.

[23] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple object access protocol (soap). Technical report, World Wide Web Consortium, 2000.

[24] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *Int. J. Softw. Tools Technol. Transf.*, 7(3):212–232, 2005.

[25] Giuseppe Castagna, Nils Gesbert, and Luca Padovani. A theory of contracts for web services. In *POPL*, pages 261–272, 2008.

[26] Stefano Ceri, Florian Daniel, Maristella Matera, and Federico Michele Facca. Model-driven development of context-aware web applications. *ACM Trans. Internet Techn.*, 7(1), 2007.

[27] Ba Cheikh, Aurélio Carrero Marcos, Halfeld-Ferrari Mirian, and Musicante Martin Alejandro. PEWS: A New Language for Building Web Service Interfaces. *J. UCS*, 11(7):1215–1233, 2005.

[28] Ba Cheikh, Halfeld-Ferrari Mirian, and Musicante Martin Alejandro. Composing Web Services with PEWS: A Trace-Theoretical Approach. In *ECOWS*, pages 65–74, 2006.

[29] Stéphanie Chollet and Philippe Lalanda. An extensible abstract service orchestration framework. In *ICWS*, pages 831–838, 2009.

[30] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language (wsdl) 1.1. Technical report, World Wide Web Consortium, 2001. DisponÃvel em http://www.w3.org/TR/wsdl.

[31] Jane Cleland-Huang, Raffaella Settimi, Xuchang Zou, and Peter Solc. The detection and classification of non-functional requirements with application to early aspects. In *RE*, pages 36–45, 2006.

[32] Standards Committee. Ieee recommended practice for software requirements specifications. *Practice*, 1998(October):37.

[33] Luiz Marcio Cysneiros, Julio Cesar Sampaio do Prado Leite, and Jaime de Melo Sabat Neto. A framework for integrating non-functional requirements into conceptual models. *Requir. Eng.*, 6(2):97–115, 2001.

[34] Andrea D'Ambrogio. A model-driven wsdl extension for describing the qos ofweb services. In *ICWS*, pages 789–796, 2006.

[35] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *ESEC / SIGSOFT FSE*, pages 109–120, 2001.

[36] María Valeria de Castro. *Aproximacíon MDA para el Desarrollo Orientado a Servicios de Sistemas de Informacíon Web: Del Modelo de Negocio al Modelo de Composicíon de Servicios Web*. PhD thesis, Universidad Rey Juan Carlos - Escuela Técnica Superior de Ingeniería de Telecomunicación, 2007.

[37] Valeria de Castro, Esperanza Marcos, and Juan M. Vara. Applying cim-to-pim model transformations for the service-oriented development of information systems. *Information & Software Technology*, 53(1):87–105, 2011.

[38] Valeria de Castro, Esperanza Marcos, and Roel Wieringa. Towards a service-oriented mda-based approach to the alignment of business processes with it systems: From the business model to a web service composition model. *International Journal of Cooperative Information Systems*, 18(2), 2009.

[39] Plácido A. de Souza Neto, Handerson Bezerra Medeiros, and Roberto Hallais Neto. Plugin extrator para verificação de composições pews. *HOLOS*, V.3:84–106, 2012. Avaiable in «http://www2.ifrn.edu.br/ojs/index.php/HOLOS/».

[40] N. Dhyanesh, G. C. Vineel, and S. V. Raghavan. Devise: A methodology for building web services based infrastructure for collaborative enterprises. In *Proceedings of the Twelfth International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, WETICE '03, pages 12–, Washington, DC, USA, 2003. IEEE Computer Society.

[41] Vassiliki Diamadopoulou, Christos Makris, Yannis Panagis, and Evangelos Sakkopoulos. Techniques to support web service selection and consumption with qos characteristics. *J. Network and Computer Applications*, 31(2):108–130, 2008.

[42] Javier-Alfonso Espinosa-Oviedo, Genoveva Vargas-Solar, José-Luis Zechinelli-Martini, and Christine Collet. Non-functional properties and services coordination using contracts. In *IDEAS*, pages 307–310, 2009.

[43] J. Fabra, V. De Castro, P. ï¿½lvarez, and E. Marcos. Automatic execution of business process models: Exploiting the benefits of model-driven engineering approaches. *Journal of Systems and Software*, (0):–, 2011.

[44] George Feuerlicht and Sooksathit Meesathit. Towards software development methodology for web services. In *SoMeT*, pages 263–277, 2005.

[45] Hamido Fujita and Mohamed Mejri, editors. *New Trends in Software Methodologies, Tools and Techniques - Proceedings of the Fifth SoMeT 2005, September 28-30, 2005, Tokyo, Japan*, volume 129 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2005.

[46] Marie-Pierre Gervais. Towards an mda-oriented methodology. In *COMPSAC*, pages 265–270, 2002.

[47] Martin Glinz. Rethinking the notion of non-functional requirements. In *in Proceedings of the Third World Congress for Software Quality (3WCSQ'05*, pages 55–64, 2005.

[48] Jaap Gordijn and Hans Akkermans. Value based requirements engineering: Exploring innovative e-commerce ideas. *REQUIREMENTS ENGINEERING JOURNAL*, 8:114–134, 2002.

[49] ATLAS Group. Atl: Atlas transformation language. Technical report, ATLAS Group, LINA & INRIA, February, 2006.

[50] Carlos Gutiérrez, David G. Rosado, and Eduardo Fernández-Medina. The practical application of a process for eliciting and designing security in web service systems. In *JISBD*, pages 143–143, 2010.

[51] Rachid Hamadi and Boualem Benatallah. A petri net-based model for web service composition. In *ADC*, pages 191–200, 2003.

[52] R. Heckel and M. Lohmann. Towards contract-based testing of web services. In Mauro Pezzé, editor, *Proceedings of the International Workshop on Test and Analysis of Component Based Systems (TACoS 2004)*, volume 116, pages 145–156, 2005.

[53] IEEE. Ieee standard computer dictionary. a compilation of ieee standard computer glossaries. *IEEE Std 610*, 1991.

[54] Ariba Inc., IBM Corp., , and Microsoft Corp. Universal description, discovery, and integration (uddi). Technical report, UDDI.org, 2000.

[55] Espinosa-Oviedo Javier-Alfonso, Vargas-Solar Genoveva, Zechinelli-Martini José-Luis, and Collet Christine. Policy driven services coordination for building social networks based applications. In *In Proc. of the 8th Int. Conference on Services Computing (SCC'11), Work-in-Progress Track*, Washington, DC, USA, July 2011. IEEE.

[56] Buhwan Jeong, Hyunbo Cho, and Choonghyun Lee. On the functional quality of service (fqos) to discover and compose interoperable web services. *Expert Syst. Appl.*, 36(3):5411–5418, 2009.

[57] Barbara A. Kitchenham, Hiyam Al-Kilidar, Muhammad Ali Babar, Mike Berry, Karl Cox, Jacky Keung, Felicia Kurniawati, Mark Staples, He Zhang, and Liming Zhu. Evaluating guidelines for reporting empirical software engineering studies. *Empirical Software Engineering*, 13(1):97–121, 2008.

[58] Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. How the design of jml accomodates both runtime assertion checking and formal verification. In *FMCO*, pages 262–284, 2002.

[59] Dewi Mairiza, Didar Zowghi, and Nur Nurmuliani. An investigation into the notion of non-functional requirements. In *SAC*, pages 311–317, 2010.

[60] D Martin, M Burstein, J Hobbs, O Lassila, D McDermott, S McIlraith, S Narayanan, M Paolucci, B Parsia, T Payne, et al. Owl-s: Semantic markup for web services. W3C Member Submission 22, 2004.

[61] Reginaldo Mendes, Paulo F. Pires, Flávia Coimbra Delicato, and Thaís Vasconcelos Batista. Webflowah: an environment for ad-hoc specification and execution of web services-based processes. In *SAC*, pages 692–693, 2009.

[62] Nikola Milanovic. Contract-based web service composition framework with correctness guarantees. In *ISAS*, pages 52–67, 2005.

[63] Nikola Milanovic. *Contract-based Web Service Composition.* PhD thesis, Humboldt-Universitat - Berlin, 2006.

[64] Nikola Milanovic. Service engineering design patterns. In *SOSE*, pages 19–26, 2006.

[65] Nikola Milanovic and Miroslaw Malek. Architectural support for automatic service composition. In *IEEE SCC*, pages 133–140, 2005.

[66] Nikola Milanovic and Miroslaw Malek. Search strategies for automatic web service composition. *Int. J. Web Service Res.*, 3(2):1–32, 2006.

[67] J. Miller and J. Mukerji. Mda guide. 2003.

[68] Giuseppe Di Modica, Orazio Tomarchio, and Lorenzo Vita. Dynamic slas management in service oriented environments. *Journal of Systems and Software*, 82(5):759–771, 2009.

[69] Ramakanta Mohanty, V. Ravi, and Manas Ranjan Patra. Web-services classification using intelligent techniques. *Expert Syst. Appl.*, 37(7):5484–5490, 2010.

[70] M. Musicante and E. Potrich. Expressing workflow patterns for web services: The case of PEWS. 12(9), september 2006.

[71] Martin A. Musicante, Edinardo Potrich, and Marcos Aurélio Carrero. A programming environment for web services. In *SAC*, pages 2363–2367, 2008.

[72] Jonathan Musset, Etienne Juliot, and Stéphane Lacrampe. Acceleo référence. Technical report, Obeo et Acceleo, 2006.

[73] Eila Ovaska, Antti Evesti, Katja Henttonen, Marko Palviainen, and Pekka Aho. Knowledge based quality-driven architecture design and evaluation. *Information & Software Technology*, 52(6):577–601, 2010.

[74] M. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-Oriented Computing: State of the Art and Research Challenges. *IEEE Computer*, 40(11), 2007.

[75] Mike P. Papazoglou. Service-Oriented Computing: Concepts, Characteristics and Directions. In *WISE*, pages 3–12, 2003.

[76] Mike P. Papazoglou, Klaus Pohl, Michael Parkin, and Andreas Metzger, editors. *Service Research Challenges and Solutions for the Future Internet - S-Cube - Towards Engineering, Managing and Adapting Service-Based Systems*, volume 6500 of *Lecture Notes in Computer Science*. Springer, 2010.

[77] Mike P. Papazoglou and Willem-Jan van den Heuvel. Service-oriented design and development methodology. *Int. J. Web Eng. Technol.*, 2(4):412–442, 2006.

[78] Jose Luis Pastrana, Ernesto Pimentel, and Miguel Katrib. Qos-enabled and self-adaptive connectors for web services composition and coordination. *Computer Languages, Systems & Structures*, 37(1):2–23, 2011.

[79] Paulo F. Pires, Mario R. F. Benevides, and Marta Mattoso. Building reliable web services compositions. In *Web, Web-Services, and Database Systems*, pages 59–72, 2002.

[80] Souza Neto Plácido A., Musicante Martin Alejandro, Vargas-Solar Genoveva, and Zechinelli-Martini José-Luis. Adding Contracts to a Web Service Composition Language. *LTPD - 4th Workshop on Languages and Tools for Multithreaded, Parallel and Distributed Programming*, September 2010.

[81] Alberto Portilla, Tan Hanh, and Javier-Alfonso Espinosa-Oviedo. Building reliable mobile services based applications. In *ICDE Workshops*, pages 121–128, 2008.

[82] Ervin Ramollari, Dimitris Dranidis, and Anthony J. H. Simons. A survey of service oriented development methodologies.

[83] Gwen Salaün, Lucas Bordeaux, and Marco Schaerf. Describing and reasoning on web services using process algebra. In *Proceeding of the 2nd International Conference on Web Services, IEEE*, 2004.

[84] Benjamin Schmeling, Anis Charfi, and Mira Mezini. Composing non-functional concerns in composite web services. In *ICWS*, pages 331–338, 2011.

[85] Ian Sommerville. *Software Engineering 6th Edition*. Addison Wesley, 2008.

[86] Andrew Stellman and Jennifer Greene. *Applied software project management*. O'Reilly, 2005.

[87] S. Thatte. Xlang: Web services for business process design. Technical report, 2001.

[88] Dirk Thißen and Pimjai Wesnarat. Considering qos aspects in web service composition. In *ISCC*, pages 371–377, 2006.

[89] Rachatrin Tongrungrojana and David Lowe. Wied: A web modelling language for modelling architectural-level information flows. *J. Digit. Inf.*, 5(2), 2004.

[90] Anargyros Tsadimas, Mara Nikolaidou, and Dimosthenis Anagnostopoulos. Extending sysml to explore non-functional requirements: the case of information system design. In *SAC*, pages 1057–1062, 2012.

[91] W. M. P. van der Aalst. Don't go with the flow: Web services compositions standards exposed. *Issue of IEEE Inteligent System*, Jan/Feb 2003.

[92] Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, Bartek Kiepuszewski, and Alistair P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[93] H. M. W. Verbeek and W. M. P. van der Aalst. Analyzing BPEL processes using Petri nets. In D. Marinescu, editor, *Proceedings of the Second International Workshop on Applications of Petri Nets to Coordination, Workflow and Business Process Management*, pages 59–78, Miami, Florida, USA, 2005. Florida International University.

[94] A. Watson. A brief history of MDA, 2008.

[95] Hua Xiao, Brian Chan, Ying Zou, Jay W. Benayon, Bill O'Farrell, Elena Litani, and Jen Hawkins. A framework for verifying sla compliance in composed services. In *ICWS*, pages 457–464, 2008.

[96] Gwyduk Yeom, Taewoong Yun, and Dugki Min. Qos model and testing mechanism for quality-driven web services selection. In *Proceedings of the The Fourth IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems, and the Second International Workshop on Collaborative Computing, Integration, and Assurance (SEUS-WCCIA'06)*, pages 199–204, Washington, DC, USA, 2006. IEEE Computer Society.

[97] Xinwen Zhang, Francesco Parisi-Presicce, Ravi S. Sandhu, and Jaehong Park. Formal model and policy specification of usage control. *ACM Trans. Inf. Syst. Secur.*, 8(4):351–387, 2005.

# APPENDIX A – Source Selection and Analysis Method: Non-Functional Requirements for Service-Based Applications

We selected the sources proposed by [57] for searching primary studies. These sources contain the works published in journals, conferences and workshops which are of recognized quality within the research community. The search for bibliography was performed in the engines are: *(i) IEEE Computer; (ii) ACM Digital Library;* and *(iii) Science Direct*. For each of the the selected sources, we used the following search query criteria:

```
(((“non functional properties”) OR (“non functional
requirements”)) AND “web service” AND “composition”))
```

After define the sources' selection, we identify those works that provided direct evidence with regard to the research questions. Deciding for the inclusion and exclusion criteria for filtering the corpus works selection, we selected those related to non-functional requirements/properties, and quality for web service based applications. Initially, the selection criteria were interpreted liberally and clear exclusions were only made with regard to title, abstract and introduction.

Based on the guidelines mentioned in [57], we established a three-step with different selection criteria:

- Step 1 - the search string must be run on the selected search engine. An initial set of studies was obtained by filtering of title, abstract, and if necessary, introduction. All the studies were selected according to the inclusion and exclusion criteria. Studies which were not clearly related to any aspect of the research questions were excluded.

| | IEEE Explorer | ACM Library | Science Direct | Total |
|---|---|---|---|---|
| Total results | 65 | 271 (75 [1]) | 166 | 502 (306 [2]) |
| Step one - results selected | 19 | 10 | 20 | 49 |
| Step one - results selected (%) | 29.23% | 13.33% | 12% | 16% |
| Step two - results selected | 7 | 3 | 9 | 19 |
| Step two - results selected (%) | 10.76% | 4% | 5.42% | 6.20% |

Table 8: Summary of the studies selected at each step.

- Step 2, the exclusion criteria were based on the following practical issues: non-English papers, non- International Conference papers and non-International Workshop papers. Specifically in the case of ACM library, we considered only the transaction journal works.
- Step 3, the papers selection process was based on detailed research questions ($RQ_1$ to $RQ_7$).

The information for each step was collected considering the 3 searchers and the the query presented previously. the results of each step were: *(i)* for each source a list of all the studies that fulfilled the query; *(ii)* a list of studies for each source which contained all the works that did not fulfill the second stage inclusion criteria; and *(iii)* the last step produced a list of works for each source which contained all the studies that fulfilled the second step (table 8).

The extraction of information was based on the research questions, and each work extraction question included the following items: *(i)* where the paper was found; *(ii)* identification of the title and main subjects; *(iii)* summary of the research; *(iv)* inclusion and exclusion criteria; *(v)* objective and result; and *(vi)* subjective results.

Table 8 shows a summary of the studies selected in each stage of the selection procedure for each source. The "Total results" were obtained by running the search string on the selected sources. The next four rows show the results obtained after applying stages one (2 first rows) and two (2 last rows) of the studies selection procedure.

In the first step, respecting the filters described, the 65 articles collected from IEEE, only 29.23% of them were in accordance with the criteria described early, representing 19 articles. In ACM Library, from 75 works collected, only 13.33% passed in the first stage filter, representing 10 articles. In Science Direct had the lowest percentage, totaling 20 of the 166 articles collected by the query, thus representing 12% of the total. Despite being the lowest relative value, the Science Direct had the largest absolute result, with 20 works in the first step. In the second stage, the percentage dropped further, and the relevant works and with accordance to the criteria have been collected as the final result. The results were respectively, 10.76%, 4% and 5.42% of total

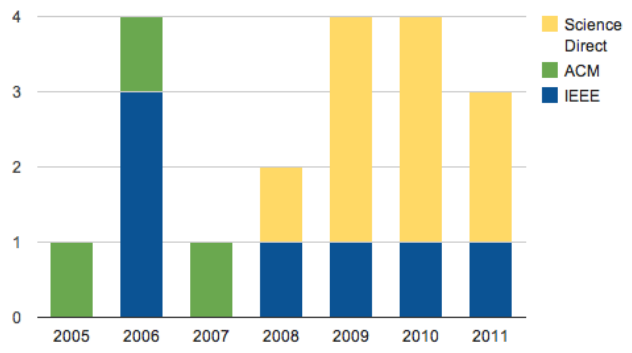


Figure 81: Publications per year.

from the IEEE, ACM and Science Direct. The highest percentage was among the works from IEEE, while the largest number od results, in absolute terms, was collected from Science Direct. The approaches resulting from this last stage were studied in depth and information concerning the detailed research questions and other fields of the extraction forms was extracted from each eork we selected. 49 works were selected in the first stage, and, only 19 works in the second stage. It represents 6.20% of the total amount of works.

Figure 81 shows the publications per year, from 2005 to 2011. 12 of 19 articles were selected in the systematic review published in 2006, 2009 and 2010, being four in each year and 6 in Science Direct source, 5 in IEEE, and only 1 at the ACM. All nine of Science Direct were published in the last 4 years. Figure also shows that the number of publications that consider classification of NFR once again increased from 2008.

# *APPENDIX B – Service-Based Non-Functional Requirement Concepts*

- NON-FUNCTIONAL (NF) ATTRIBUTE - An attribute that describes the quality or characteristics of a functional requirement. For example *confidentiality* and *privacy* may be non-functional attributes for the functional requirement *user registration*.
- NON-FUNCTIONAL (NF) REQUIREMENT - A group of semantically correlated non-functional attributes (NFA). For example, *security* is an NF Requirement that comprises attributes such as *confidentiality* and *integrity*.
- CONSTRAINT - A constraint prevents the system from achieving more than its goal. With the definition of constraints, the system can be more robust, and unexpected problems can be solved before they happen. For example, in a banking system, the customer can only withdraw money if they have positive balance in the account.
- CONSTRAINT TYPE - Represents the types of constraints that could be expressed, the types are: business and data (*value) constraints. When modeling a system requirement the analyst can identify if there are restrictions on business, or data, or both.
- CONTRACT - Is the formalization of obligations (requires) and benefits (ensures) of a function, service activity or component. The following questions can be used to define contracts: *What does it expect? What does it guarantee?* Contract can be crucial to software correctness that they should be part of the design process.
- EXCEPTIONAL BEHAVIOUR - Are alternative execution paths if any condition or restriction is not respected. For example, if a user's password is not correct after three attempts, the user's account is locked for security.
- POLICY - A policy is a set of rules applied to a particular scope. This scope can be defined as an action, an activity, a function or a workflow. A policy is a composition of contracts

applied to a non-functional application requirement. For example, a security policy of a system constraint includes authentication, access, data privacy, and so on.

- USE CASE - Represents a behavior that can be executed in order to realize (parts of) a functional requirement.
- REQUIREMENT - A requirement is a super type for functional and non-functional requirements. Thus, the use cases can be related to both types of requirements.
- SERVICE ACTIVITY - Represents a function of a software system or its component that are implemented through services. A Service Activity may be calculations, technical details, data manipulation and processing and other specific functionality that are available to be accessed on the Internet.

# *APPENDIX C – π-PEWS Language*

$\pi$-PEWS specifies assertions that are checked during the execution of a program. These assertions describe pre- and post-conditions for a given operation or compound service. In the case these conditions are not verified, the contract defines correcting actions (described as a PEWS path expression). The grammar defining the new language is defined as follows: A $\pi$-PEWS program is similar to a PEWS program, but with the possibility of adding contract definitions at the end of the program. Path expressions, defining the service workflow, are described by the *Service* non-terminal of the grammar below. Since contracts and workflow are separate concerns, the syntax of path expressions remain unchanged, from the original version of the language.

(1) *Program* ::= $[\![$ (“**var**” **id** “=”$ArithExp$)* (“**service**” **id** “=”$Service$)*$Service\ Contract^*$ $]\!]$

(2) *Service* ::= $[\![$ **id** $]\!]$ | $[\![$ $Service$ “.” $Service$ $]\!]$ | $[\![$ $Service$“+” $Service$ $]\!]$ | $[\![$ $Service$“$\|$” $Service$ $]\!]$
| $[\![$ $Service$“$*$” $]\!]$ | $[\![$ “[” $BooleanCondition$ “]” $Service$ $]\!]$ | $[\![$ “{” $Service$ “}” $]\!]$

The definition of contracts is given below. It includes a name for the contract, as well as its four component sections:

- Target service of the contract: This section specifies the service to which the contract applies. This can be an operation or a compound service (which also defines a scope for the contract).
- Preconditions and their actions. This section defines the assertions that will be verified before the target service is executed.
- Post-conditions and their actions. This section defines the assertions that will be verified at the end of the target service execution.
- Time constraints for the services on the contract scope.

(3) *Contract* ::= $[\![$ “**def**” “**Contract**” *Id* “**{**” *IsAppliedTo Requires* Ensures* (TimeConstraint)?* “**}**” $]\!]$

The directive "**isAppliedTo:**" defines the target service and scope of the contract. This service is given by the identifier appearing next to the keyword. The target service can be a simple operation or a compound service. In the former case, the contract cannot contain any time restriction. In the case of a contract defined for a compound service, the operations and services that form the contract can participate of the time constraint expressions defined by the contract.

(4) *isAppliedTo* ::= ⟦ "**isAppliedTo**" "**:**" *ident* "**;**" ⟧

The "**requires**" and "**ensures**" parts of a contract define the pre-conditions (resp. post-conditions) to be checked for each contract. In the case of pre-conditions, they will be verified before the service is executed. Post-conditions will be checked after the execution of the service. In both cases, when the assertion fails, the associated action will be executed. Actions are defined as services, and written as PEWS path expressions.

(5) *Requires* ::= ⟦ "**requires**" *BooleanCondition ( "(" onFailureDo ")" )?* "**;**" ⟧

(6) *ensures* ::= ⟦ **'ensures'** *BooleanCondition ( "(" onFailureDo ")" )?* "**;**" ⟧

(7) *onFailureDo* ::= ⟦ **'onFailureDo'** *Service* ⟧

Time Constraints are defined as relational expressions build from the operators.

(8) *timeConstraint* ::= ⟦ **'timeConstraint'** **':'** (⟦ "**meet**" ⟧ | ⟦ "**start**" ⟧ | ⟦ "**overlap**" ⟧)
**'('** (*opName* | *timeConstraint*) **','** (*opName* | *timeConstraint*) **')'** ⟧

The model of temporal relations proposed here follows the main ideas presented in [8, 18]. The model is used to impose additional restrictions to the order in which operations of a web service are performed at execution time. For example, the confirmation of the bank authorization request must arrive at most one minute after the service call.

A service composition time relation is represented by the following model: Let $\Omega$ be a set of web services ($\omega_1$, . . ., $\omega_n$) to be composed and $\Theta$, a set of temporal relations $r_t$ that can be applied to $\Omega$. Temporal relations are defined by predicates $r_t$ : $\Omega$ x $\Omega$, where $r_t \in$ { ***meet, overlap, start*** }.

For example, the expression ***meet(***$\boldsymbol{\omega_1, \omega_2}$***)*** states that the execution of operation $\omega_2$ will begin as soon as $\omega_1$ finishes. The temporal relations are described below:

***Meet.*** The restriction specified by ***meet(***$\boldsymbol{\omega_1, \omega_2}$***)*** specifies that $\omega_2$ will be executed immediately after $\omega_1$ finishes.

***Overlap.*** The constraint given by ***overlap(*$\boldsymbol{\omega_1, \omega_2}$*)*** states that execution of the two services overlap in time. The restriction also states that the service $\omega_1$ initiates before $\omega_2$.

***Start.*** The specification of ***start(*$\boldsymbol{\omega_1, \omega_2}$*)*** states that the two services have their execution in parallel and that they start at the same time.

# *APPENDIX D – To Publish Music Case Study Diagrams*

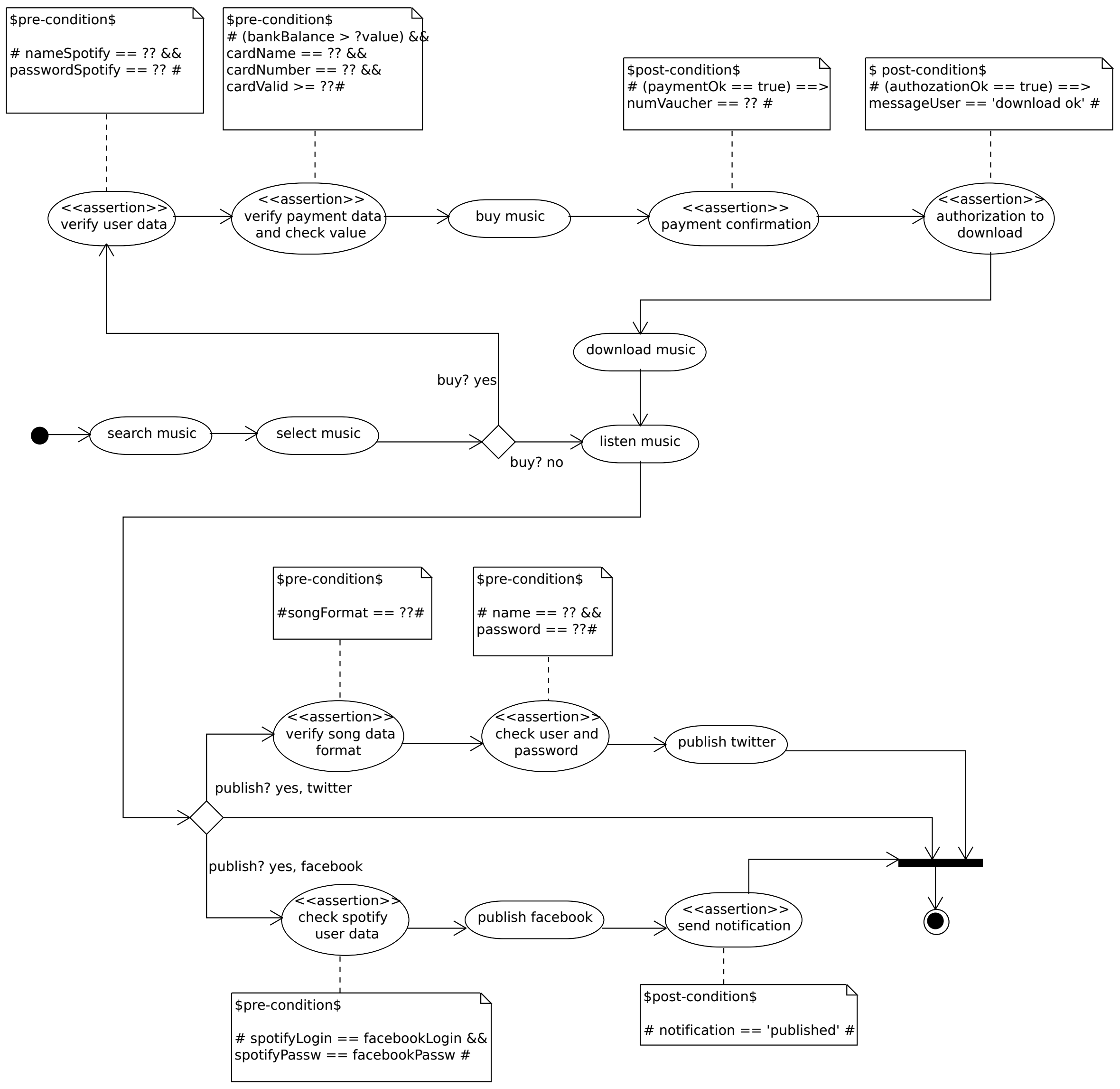

$pre-condition$
# nameSpotify == ?? && passwordSpotify == ?? #
$pre-condition$
# (bankBalance > ?value) && cardName == ?? && cardNumber == ?? && cardValid >= ??#
$post-condition$
# (paymentOk == true) ==> numVaucher == ?? #
$ post-condition$
# (authozationOk == true) ==> messageUser == 'download ok' #
<<assertion>> verify user data
<<assertion>> verify payment data and check value
buy music
<<assertion>> payment confirmation
<<assertion>> authorization to download
download music
buy? yes
search music
select music
listen music
buy? no
$pre-condition$
#songFormat == ??#
$pre-condition$
# name == ?? && password == ??#
<<assertion>> verify song data format
<<assertion>> check user and password
publish twitter
publish? yes, twitter
publish? yes, facebook
<<assertion>> check spotify user data
publish facebook
<<assertion>> send notification
$pre-condition$
# spotifyLogin == facebookLogin && spotifyPassw == facebookPassw #
$post-condition$
# notification == 'published' #

Figure 82: $\pi$-*ServiceProcess* - To Publish Music.

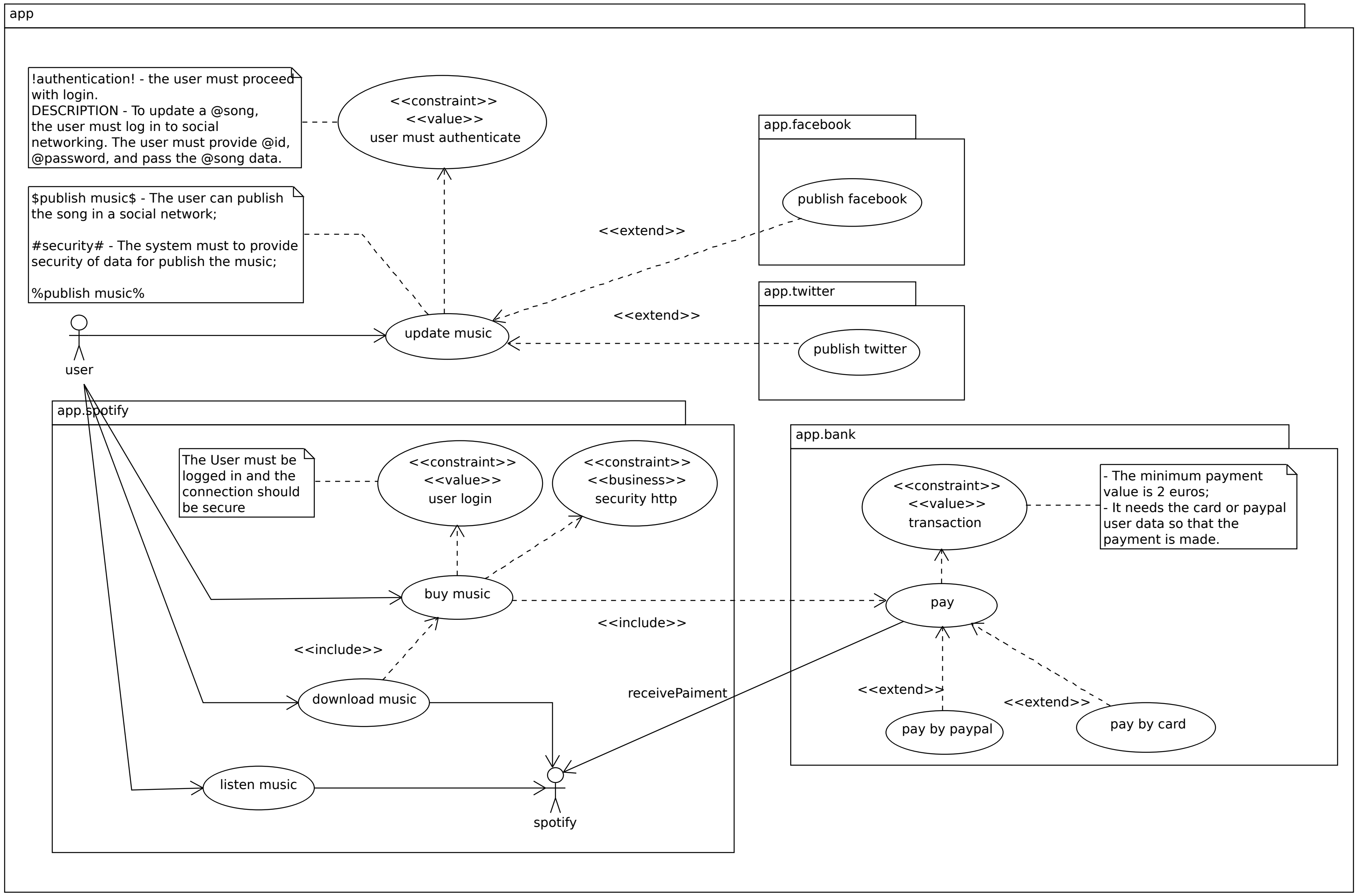
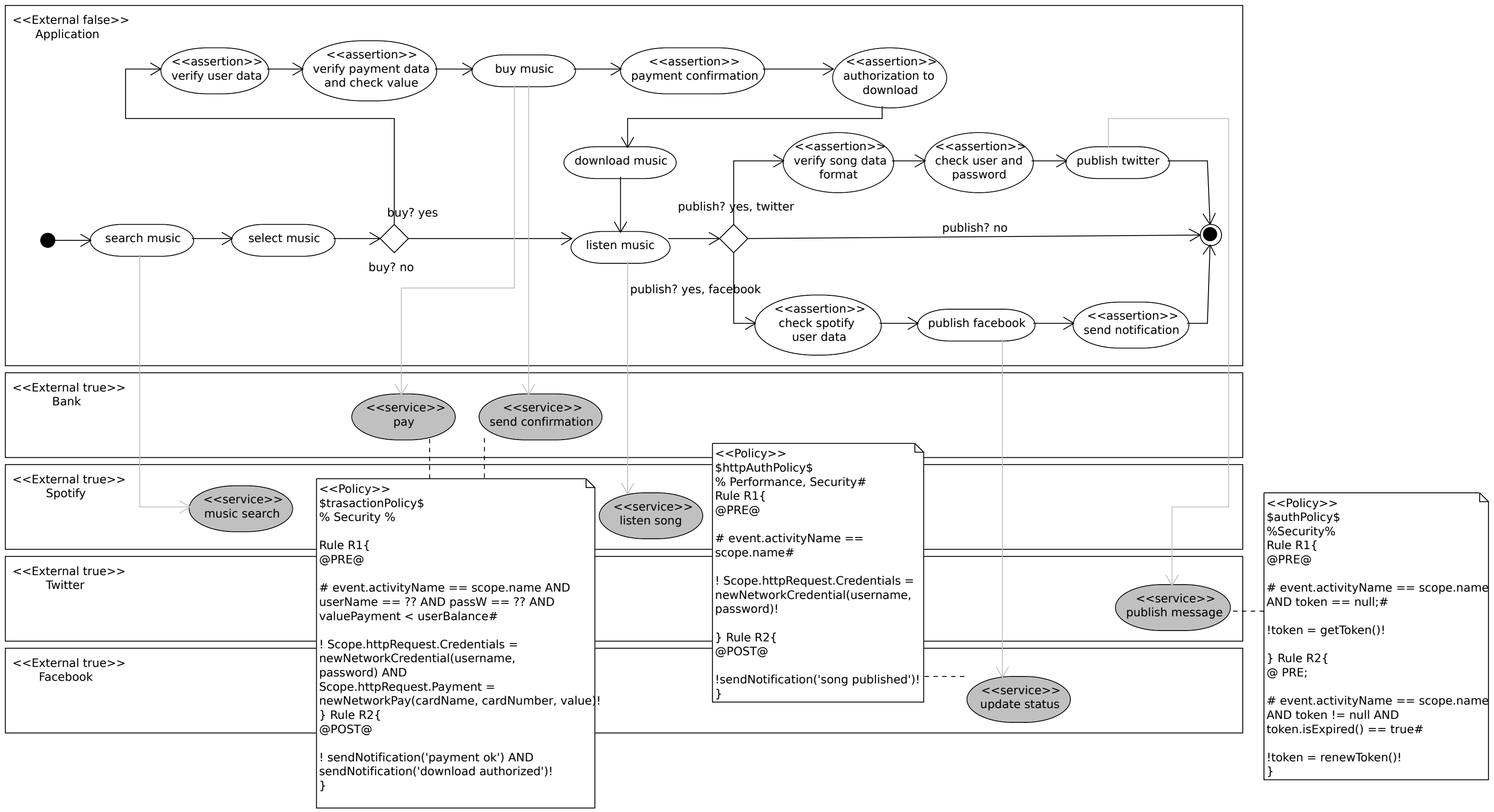

app
!authentication! - the user must proceed with login.
DESCRIPTION - To update a @song, the user must log in to social networking. The user must provide @id, @password, and pass the @song data.
<<constraint>> <<value>> user must authenticate
$publish music$ - The user can publish the song in a social network;
#security# - The system must to provide security of data for publish the music;
%publish music%
app.facebook
publish facebook
<<extend>>
app.twitter
publish twitter
<<extend>>
user
update music
app.spotify
The User must be logged in and the connection should be secure
<<constraint>> <<value>> user login
<<constraint>> <<business>> security http
buy music
<<include>>
<<include>>
download music
listen music
receivePaiment
spotify
app.bank
<<constraint>> <<value>> transaction
- The minimum payment value is 2 euros;
- It needs the card or paypal user data so that the payment is made.
pay
<<extend>>
<<extend>>
pay by paypal
pay by card

Figure 83: $\pi$-*UseCase* - To Publish Music.


<<External false>>
Application
<<assertion>> verify user data
<<assertion>> verify payment data and check value
buy music
<<assertion>> payment confirmation
<<assertion>> authorization to download
download music
<<assertion>> verify song data format
<<assertion>> check user and password
publish twitter
search music
select music
buy? yes
buy? no
listen music
publish? yes, twitter
publish? no
publish? yes, facebook
<<assertion>> check spotify user data
publish facebook
<<assertion>> send notification
<<External true>>
Bank
<<service>> pay
<<service>> send confirmation
<<External true>>
Spotify
<<service>> music search
<<service>> listen song
<<External true>>
Twitter
<<service>> publish message
<<External true>>
Facebook
<<service>> update status

```
<<Policy>>
$trasactionPolicy$
% Security %

Rule R1{
@PRE@

# event.activityName == scope.name AND
userName == ?? AND passW == ?? AND
valuePayment < userBalance#

! Scope.httpRequest.Credentials =
newNetworkCredential(username,
password) AND
Scope.httpRequest.Payment =
newNetworkPay(cardName, cardNumber, value)!
} Rule R2{
@POST@

! sendNotification('payment ok') AND
sendNotification('download authorized')!
}
```

```
<<Policy>>
$httpAuthPolicy$
% Performance, Security#
Rule R1{
@PRE@

# event.activityName ==
scope.name#

! Scope.httpRequest.Credentials =
newNetworkCredential(username,
password)!

} Rule R2{
@POST@

!sendNotification('song published')!
}
```

```
<<Policy>>
$authPolicy$
%Security%
Rule R1{
@PRE@

# event.activityName == scope.name
AND token == null;#

!token = getToken()!

} Rule R2{
@ PRE;

# event.activityName == scope.name
AND token != null AND
token.isExpired() == true#

!token = renewToken()!
}
```

Figure 84: $\pi$-*ServiceComposition* - To Publish Music.