POLITECNICO DI MILANO

COMO CAMPUS

Engineering School

Master of Science in Computer Engineering for Communication

Master Thesis

Generating Application Independent Test Cases from Business Process Models

Supervisors:

Piero Fraternali

Massimo Tisi

Candidate:

María Luz Racca

Student Id: 736391

ACADEMIC YEAR 2009-2010

Abstract

Business process models are a good tool to depict how is the flow and interaction among tasks inside a company and who is the responsible of doing each one of them. In the software engineering field, these models are used today as the first step to derive automatically software applications. The focus of the present work is to extract all the possible scenarios present on a business process model in order to test and evaluate these generated applications. Hence, the result of this work consists in a platform independent model representing a test suite where each test case will correspond to each path of the business process. Besides, as the parameters of the process can take different values depending on the flow or the actions done by the different users, the model will contemplate the insertion of test data for each parameter of the process. Finally, this report will analyze existing tools, give a potential solution to the existing problem and give advices for further studies or optimization.

Sommario

I modelli di processi aziendali sono un ottimo strumento per descrivere come è il flusso e l'interazione fra tutte le attività all'interno di una azienda e chi è il responsabile di fare ognune di loro. Nel campo dell'ingegneria del software, questi modelli sono utilizzati oggi come il primo passo per ottenere automaticamente le applicazioni di software. L'obiettivo del presente lavoro è quello di estrarre tutti i possibili scenari presenti su un modello di processo aziendale, al fine di verificare e valutare queste applicazioni generate. Quindi, il risultato di questo lavoro consiste in un modello indipendente della piattaforma usata, che rappresenta una serie di casi di prova, dove ognuno corrisponderà a un flusso diverso del processo aziendale. Inoltre, come i parametri del processo possono assumere valori diversi, secondo le azione svolte dai vari utenti, il modello verrà contemplare l'inserimento dei dati di prova per ogni parametro del processo. Infine, questo lavoro analizza ache gli strumenti esistenti, da una possibile soluzione per il problema esistente e anche consigli per ulteriori studi o ottimizzazioni.

Contents

List of Figures

Chapter 1

Introduction

To begin with, the growing complexity of software systems has produced an urgent need to find tools to facilitate the complete or partial generation of the system in an automated way. As a result, companies would spend less effort on building and maintaining their software applications and focusing instead in the real business matters.

On the other hand, searching for the best approach to reflect the business needs in the software system under development, business process specification languages emerged as a solution for easing the definition of the business constraints. Moreover, they facilitate the communication between managers, analysts, developers and end users.

These concepts have joined today giving, as a result, automated generation tools based on business process models. This kind of automated tools, needs definitely some method to assure that the generated application is working correctly with the passing of time. They should be strong enough to support changes, upgrades or even process re-engineering. At this point the importance of quality assurance and the existence of a solid testing process arises. As we are talking of automatic tools, the optimal solution would be to have an automatic test generating feature included. Furthermore, if the test cases are born from the same business process model, the probability of correlation between the test cases and the model becomes higher and the chance of inconsistency is lowered.

The proposed work is meant to be immersed in a automatic testing generation framework. This framework is based in a model driven engineering approach that uses models (in this case business process models) to, from one side, generate automatically the software system and, on the other side it will do the automatic generation a test suite for that system. Finally, this framework, even tough it can be applied in the future in many different environments, is suggested for being implemented into the *BPM WebRatio*.

The step of the mentioned framework that is the focus of the present work is the one that, taking as an input a business process model, generates a platform

independent test cases. The generation of these test cases starting from a business process model is not something minor given the fact that the complexity of processes inside a company is commonly high. As this happens, it is necessary to select the more important paths of the process in order to guarantee that the web application is optimally covered by the generated test cases. This work will also explain how these selections are made and which is the used criterion.

The work is organized like follows. The *Background* chapter includes all the basic information that can be needed in order to understand the context of the proposed framework. Subsequently, the *State of the art* is presented. Here, even tough there is no explicit mention of the use of business process models to derive test cases, a summary of the existing similar solutions are mentioned. The aim of the chapter is to explore different alternatives and summarizing the most important suggestions from different authors. The next chapter is the *Application Context*. As said before, the proposal is meant to be inserted into a complete automatic generation framework. Therefore, that chapter includes a description of that application context. As it will be mentioned, the proposal is meant to work also inside other contexts but, to show the current one is important to demonstrate the feasibility of the final objective. The Chapter 5 includes the basic components design of the proposal to generate the platform independent test cases. The next Chapter called *Implementation* shows a prototype using the previous exposed design. Finally, the last Chapter (*Experimental Results*) shows a complete example using BPM WebRatio as the modeling tool.

Chapter 2

Background

In this chapter, all the basic concepts covered in this work will be revised in a summarized way to facilitate the understanding of the proposal.

2.1 Testing Object Oriented Systems

2.1.1 Basic concepts

Definitions

Software Testing

Software testing is the execution of code using combinations of input and state selected to reveal bugs. Some definitions of testing include other verification and validation activities, but here testing is limited to running an implementation with input selected by test design and evaluating the response. [Bin00]

Model-Based Testing

Model-Based testing is a recurrent subject when talking about test case automation. The main reason is that the complexity of software requires the development of models to understand what is the system's behavior. Understanding the system's behavior is the first thing to do before starting to think about the creation of a test case that is suitable for that system.

> A model has four main elements: subject, point of view/theory, representation, and technique.
>
> - *Subject.* A model must have a well defined subject. For example, models of airframes, buildings, forest fires, bridges, weather, and eco-

nomics have been developed. In testing, we are interested in models of the IUT[1] that will help us select effective test cases.

- *Point of view / Theory.* A model must be based on a frame of reference and principles that can guide identification of relevant issues and information. Software testing models typically express required behavior and focus on aspects of structure or elements suspected to be buggy. They must establish the information necessary to produce and evaluate test cases.
- *Representation.* A modeling technique must have a means to express a particular model. This may be a wire frame image for a CAD miodel of an automobile body, a blueprint for a building, or equations for a mathematical model. Many software testing models use graphs[2].
- *Technique.* Models are complex artifacts. The skill and artisanship of the modeler matters. Some modeling approaches have an extensive literature covering style and heuristics. Other approaches are more esoteric.

[Bin00].

Testability

A testable model has sufficient information to allow automatic generation of test cases. We can devise and program an algorithm that will produce ready-to-run test cases with only the information in the model. This requires a model that meets the following criteria:

- It is a complete and accurate reflection of the kind of implementations to be tested. The model must represent all features that should be tested.
- It allows abstraction of detail, which, if modeled explicitly, would make the cost of testing prohibitive.
- It preserves detail that is essential for revealing faults and demonstrating conformance.
- It represents all events to which the IUT (Implementation Under Test) must respond. A message, an exception, or interrupt can be produced to test each event.

[1] Implementation Under Test: The implementation that is the subject of test design or is being exercised by a test suite

[2] The graphs used in these test models are simple circle and arrow drawings.

- It represents all actions such that we can generate a program to decide whether some actions has been produced.
- It defines state so that the checking of resultant state can be automated.

[Bin00]

Automation

The automation of software testing can be seen as a problem of software engineering [Bin00]. As a consequence, it includes several things to have in mind. These things are show in the simplified scenario of the figure 2.1.

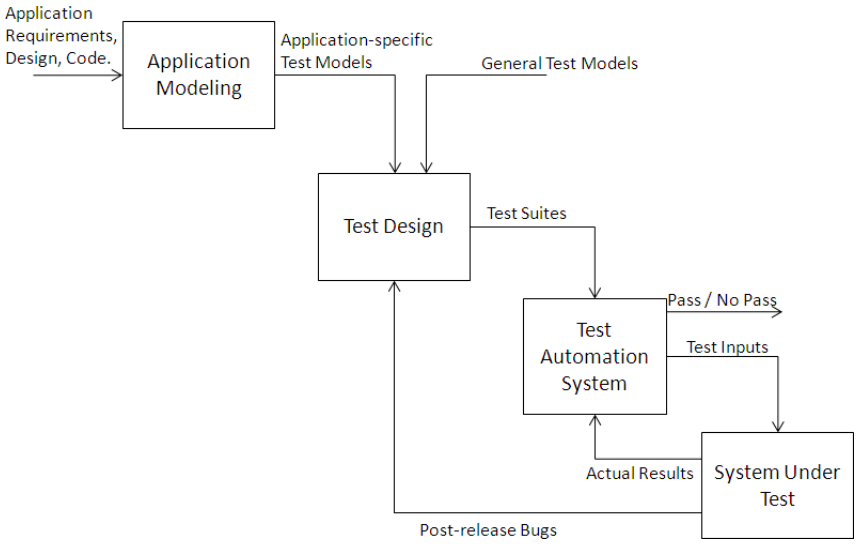

Figure 2.1: A systems engineering view of testing [Bin00].

This scenario begins with the *application models* which are developed to represent the required behavior of the system. These models can be derived directly into *application-specific Test Models* or can also be generated manually. On the other hand, *General Test Models* offer a systematic and repeatable means to generate test suites. Both kind of models are combined to generate the *Test Design* that consists basically in an algorithm taking those models and producing the application test suite as an output.

Afterwards, the *Test Automation System* receives as input the generated test suite and evaluates the *System Under Test*. This automated system typically will start the implementation under test, set up its environment, bring it to the required pretest state, apply the test inputs, and evaluate the resulting output and state.

This work is based mainly in the automation of the *Test Design* step, involving the following matters:

1. Receiving as input the *Application-Specific Models* (That will enter into the form of an *Business Process Diagram*)
2. Apply the knowledge acquired from *General Test Models*.
3. Design an algorithm to convert the models into *Test Cases* conforming a *Test Suite*

2.1.2 Coverage Criteria

The *coverage* criteria defines an amount of how much complete a test suite is. In other words, how many cases the test suite is covering.

The stronger the coverage criterion the more paths have to be selected. Below is a list of the most cited criteria.

- **Statement coverage**: Is the percentage of all source code statements executed at least once by a test suite.
- **Branch coverage:** Encounter all branches in the program, e.g. the predicate of an if-statement must be evaluated to both *true* and *false*.
- **Condition coverage**: Each clause within each condition must be executed to both *true* and *false*, some time during execution.
- **Multiple-condition coverage**: Each combination of truth values of each clause of each condition must be executed during execution.
- **Path coverage**: Traverse each path in the flowgraph.

When the term coverage is used without quantification, 100 percent is usually understood.

2.2 Business Process Modeling Notation (BPMN)

The *Business Process Modeling Notation* has been born to simplify the way of writing and understanding business process.

The standard specification of the BPMN is issued by the OMG (Object Management Group).

Founded in 1989, the *Object Management Group*, Inc. (OMG) is an open membership, not-for-profit computer industry standards consortium that produces and

maintains computer industry specifications for interoperable, portable and reusable enterprise applications in distributed, heterogeneous environments.

Within the OMG, the *Business Process Management Initiative* (BPMI) has developed a standard *Business Process Modeling Notation* (BPMN). The primary goal of BPMN is to provide a notation that is readily understandable by all business users, from the business analysts that create the initial drafts of the processes, to the technical developers responsible for implementing the technology that will perform those processes, and finally, to the business people who will manage and monitor those processes. Thus, BPMN creates a standardized bridge for the gap between the business process design and process implementation.

The *BPMN* specification defines the notation and semantics of a *Business Process Diagram* (BPD) and represents the amalgamation of best practices within the business modeling community. The intent of BPMN is to standardize a business process modeling notation in the face of many different modeling notations and viewpoints. In doing so, BPMN will provide a simple means of communicating process information to other business users, process implementers, customers, and suppliers.

2.2.1 BPMN basics

The *Business Process Diagram* is done as a composition of graphical objects. The basic set of objects, defined in [OMG09b], is the one described below:

Flow Objects

Flow Objects are the main graphical elements to define the behavior of a Business Process. There are three Flow Objects:

- *Events:* An event is something that happens during the course of a business process.
- *Activities:* An activity is a generic term for work that company performs.
- *Gateways:* A Gateway is used to control the divergence and convergence of Sequence Flow.

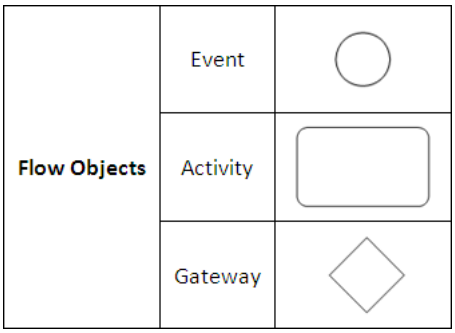

Figure 2.2: Graphical representation of Flow Objects using BPMN.

Connecting Objects

There are three ways of connecting the Flow Objects to each other or other information:

- *Sequence Flow:* A Sequence Flow is used to show the order that activities will be performed in a Process.
- *Message Flow:* A Message Flow is used to show the flow of messages between two participants that are prepared to send and receive them.
- *Association:* An Association is used to associate information with Flow Objects.

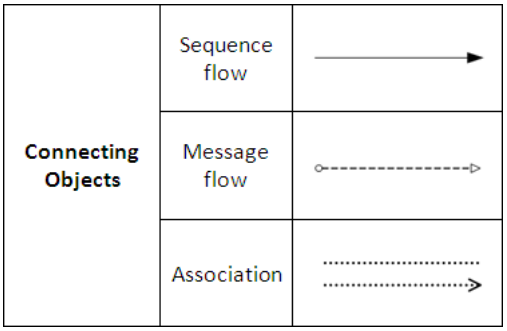

Figure 2.3: Graphical representation of Connecting Objects using BPMN.

Swimlanes

There are two ways of grouping the primary modeling elements through:

- *Pools:* A Pool represents a Participant in a Process.

- *Lanes:* A Lane is a sub-partition within a Pool. Lanes are used to organize and categorize activities.

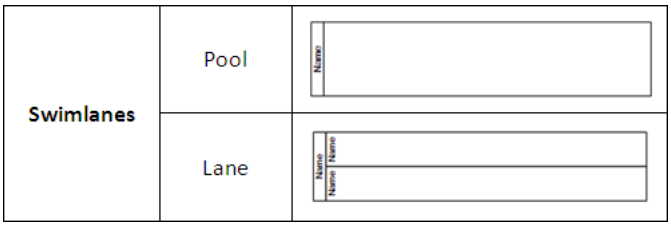

Figure 2.4: Graphical representation of Swimlanes using BPMN.

Artifacts

Artifacts are used to provide additional information about the Process. There are three standardized Artifacts, but modelers or modeling tools are free to add as many Artifacts as required. There may be additional BPMN efforts to standardize a larger set of Artifacts for general use or for vertical markets. The current set of Artifacts include:

- *Data Object:* Data Objects provide information about what activities require to be performed and/or what they produce.
- *Group:* A grouping of activities that are within the same category.
- *Annotation:* Text Annotations are a mechanism for a modeler to provide additional information for the reader of a BPMN Diagram.

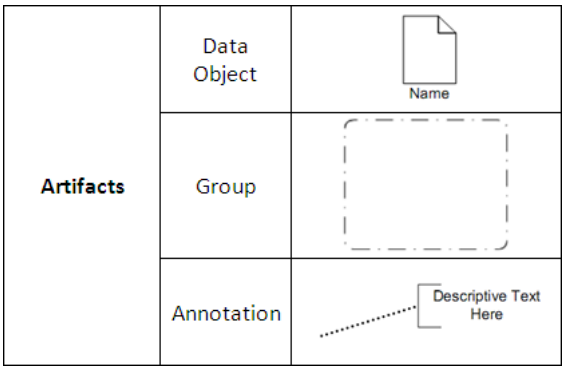

Figure 2.5: Graphical representation of the different Artifacts using BPMN.

2.2.2 Example

For example, in the Fig. 2.6 we can see a very simple business process using this notation.

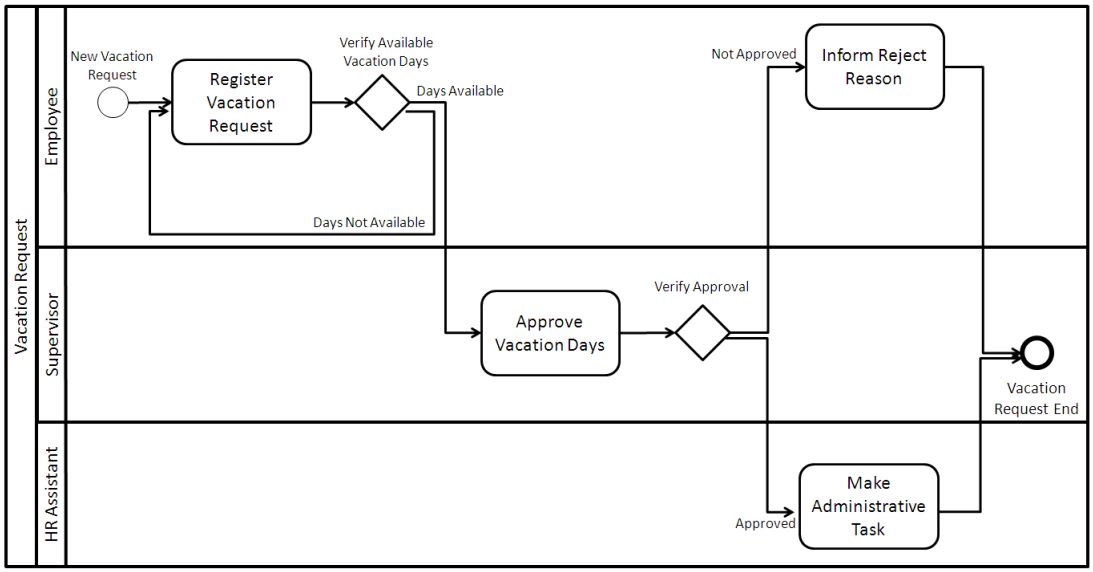

Figure 2.6: Example of a BPD (Business process Diagram).

2.2.3 The XPDL format

The basic concepts that underlie XPDL were formulated by individuals working together in the *Workflow Management Coalition* (WfMC) who were from companies developing workflow and business process management tools. These concepts were embodied in a metamodel and glossary which then guided the specification of interfaces for various aspects of the overall problem. The interchange of process definitions between different tools and also different vendors was regarded as an essential piece of this whole and the first version of a standard interchange language was WPDL, published by the WfMC in November 1998.

The growing popularity of XML and its use for defining document formats for the Internet, combined with some years of accumulated experience using WPDL in workflow and BPM tools, led to the creation of XPDL.

XPDL provides a standard graphical approach to *Business Process Definition* based on BPMN graphics. XPDL provides a standard file format for persisting BPMN diagrams and interchanging Process definitions. The file format is based on the WfMC meta-model which establishes a framework for defining, importing and exporting process definitions for numerous products including execution engines, simulators,

BPA modeling tools, Business Activity Monitoring and reporting tools. The schema defining the format is extensible and provides vendor and user extension capabilities as well as a natural path for future versions of the standard. Mappings to specific execution languages (e.g. BPEL) and other XML based specifications are possible. Finally, BPMN Model Portability conformance classes greatly increase the likelihood of true portability at the design level between a significant number of different vendor tools [Sha08].

2.3 Workflow Patterns

The *Workflow Patterns* describe different recurrent behavior of business processes. They were mainly studied by Aalst et al [AHKB03] and they are still used as a way of standardizing the behavior of *workflows* or *business processes*.

A brief summary of the identified patterns is exposed below:

2.3.1 Basic Control Flow Patterns

- **Sequence**: A task in a process in enabled after the completion of a preceding task in the same process.
- **Parallel Split**: The divergence of a branch into two or more parallel branches each of which execute concurrently.
- **Synchronization**: The convergence of two or more branches into a single subsequent branch such that the thread of control is passed to the subsequent branch when all input branches have been enabled.
- **Exclusive Choice**: The divergence of a branch into two or more branches such that when the incoming branch is enabled, the thread of control is immediately passed to precisely one of the outgoing branches based on a mechanism that can select one of the outgoing branches.
- **Simple Merge**: The convergence of two or more branches into a single subsequent branch such that each enablement of an incoming branch results in the thread of control being passed to the subsequent branch.

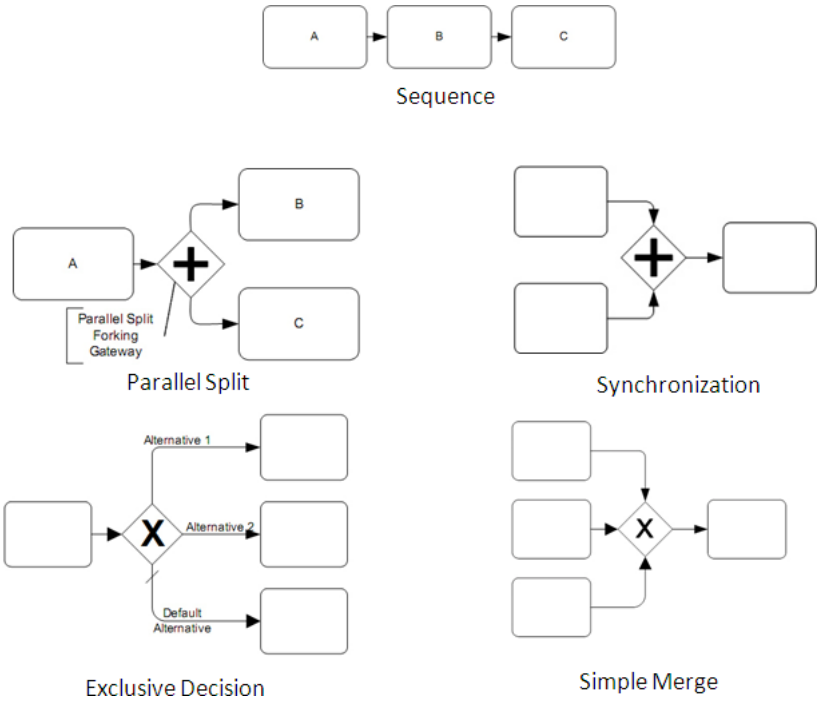

Figure 2.7: Basic Control Flow Patterns.

2.3.2 Branching Patterns

- **Multi Choice**: The divergence of a branch into two or more branches such that when the incoming branch is enabled, the thread of control is immediately passed to one or more of the outgoing branches based on a mechanism that selects one or more outgoing branches.

- **Structured Synchronizing Merge**: The convergence of two or more branches (which diverged earlier in the process at a uniquely identifiable point) into a single subsequent branch such that the thread of control is passed to the subsequent branch when each active incoming branch has been enabled. The **Structured Synchronizing Merge** occurs in a structured context, i.e. there must be a single *Multi-Choice* construct earlier in the process model with which the *Structured Synchronizing Merge* is associated and it must merge all of the branches emanating from the *Multi-Choice*. These branches must either flow from the *Structured Synchronizing Merge* without any splits or joins or they must be structured in form (i.e. balanced splits and joins).

- **Multiple Merge**: The convergence of two or more branches into a single subsequent branch such that each enablement of an incoming branch results in the

thread of control being passed to the subsequent branch.

- **Structured Discriminator**: The convergence of two or more branches into a single subsequent branch following a corresponding divergence earlier in the process model such that the thread of control is passed to the subsequent branch when the first incoming branch has been enabled. Subsequent enablement's of incoming branches do not result in the thread of control being passed on. The *Structured Discriminator* construct resets when all incoming branches have been enabled. The *Structured Discriminator* occurs in a structured context, i.e. there must be a single *Parallel Split* construct earlier in the process model with which the *Structured Discriminator* is associated and it must merge all of the branches emanating from the *Structured Discriminator*. These branches must either flow from the *Parallel Split* to the *Structured Discriminator* without any splits or joins or they must be structured in form (i.e. balanced splits and joins).

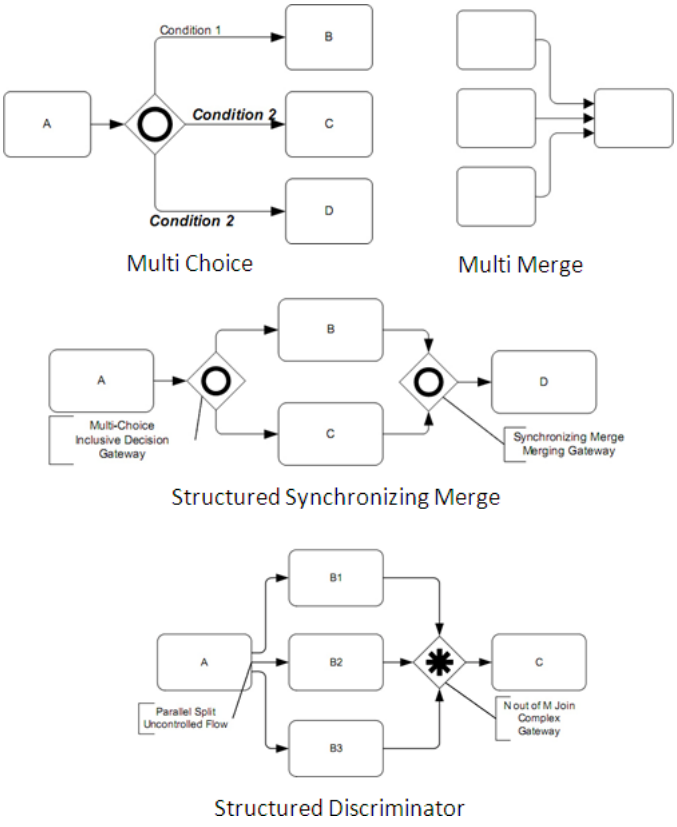

Figure 2.8: Branching Patterns.

2.3.3 Structural Patterns

- **Arbitrary Cycles**: The ability to represent cycles in a process model that have more than one entry or exit point. It must be possible for individual entry and exit points to be associated with distinct branches.
- **Implicit Termination**: A given process (or sub-process) instance should terminate when there are no remaining work items that are able to be done either now or at any time in the future and the process instance is not in deadlock. There is an objective means of determining that the process instance has successfully completed.

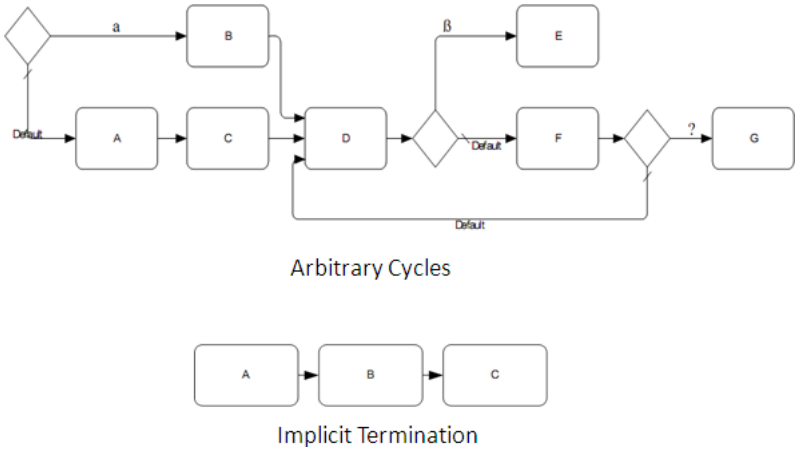

Figure 2.9: Structural Patterns.

2.4 Model Driven Architecture (MDA)

Model Driven Architecture is an industry architecture proposed by the *Object Management Group* (OMG) that addresses full life-cycle application development, data, and application integration standards that work with multiple middleware languages and interchange formats. MDA unifies some of the industry best practices in software architecture, modeling, metadata management, and software transformation technologies that allow a user to develop a modeling specification once and target multiple technology implementations by using precise transformations/mappings [Bud04].

The basic *Model Driven Architecture* principles are developed by the *OMG* and described at [MM03]:

System

We present the MDA concepts in terms of some existing or planned system. That system may include anything: a program, a single computer system, some combination of parts of different systems, a federation of systems, each under separate control, people, an enterprise, a federation of enterprises Much of the discussion focuses on software within the system.

Model

A *model* of a system is a description or specification of that system and its environment for some certain purpose. A model is often presented as a combination of drawings and text. The text may be in a modeling language or in a natural language.

Model-Driven

MDA is an approach to system development, which increases the power of models in that work. It is model-driven because it provides a means for using models to direct the course of understanding, design, construction, deployment, operation, maintenance and modification.

Architecture

The *architecture* of a system is a specification of the parts and connectors of the system and the rules for the interactions of the parts using the connectors. The *Model-Driven Architecture* prescribes certain kinds of models to be used, how those models may be prepared and the relationships of the different kinds of models.

Viewpoint

A viewpoint on a system is a technique for abstraction using a selected set of architectural concepts and structuring rules, in order to focus on particular concerns within that system. Here *abstraction* is used to mean the process of suppressing selected detail to establish a simplified model. The *Model-Driven Architecture* specifies three viewpoints on a system:

- **Computation Independent Viewpoint**: It focuses on the on the environment of the system, and the requirements for the system; the details of the structure and processing of the system are hidden or as yet undetermined.
- **Platform Independent Viewpoint**: It focuses on the operation of a system while hiding the details necessary for a particular platform. A platform independent view shows that part of the complete specification that does not change from one platform to another.

- **Platform Specific Viewpoint**: It combines the platform independent viewpoint with an additional focus on the detail of the use of a specific platform by a system.

View

A viewpoint model or view of a system is a representation of that system from the perspective of a chosen viewpoint.

Platform

A platform is a set of subsystems and technologies that provide a coherent set of functionality through interfaces and specified usage patterns, which any application supported by that platform can use without concern for the details of how the functionality provided by the platform is implemented.

2.4.1 The Computation Independent Model (CIM)

A *computation independent model* is a view of a system from the computation independent viewpoint. A CIM does not show details of the structure of systems. A CIM is sometimes called a domain model and a vocabulary that is familiar to the practitioners of the domain in question is used in its specification.

It is assumed that the primary user of the CIM, the domain practitioner, is not knowledgeable about the models or artifacts used to realize the functionality for which the requirements are articulated in the CIM. The CIM plays an important role in bridging the gap between those that are experts about the domain and its requirements on the one hand, and those that are experts of the design and construction of the artifacts that together satisfy the domain requirements, on the other.

2.4.2 Platform Independent Model (PIM)

A *platform independent model* is a view of a system from the platform independent viewpoint. A PIM exhibits a specified degree of platform independence so as to be suitable for use with a number of different platforms of similar type. A very common technique for achieving platform independence is to target a system model for a technology-neutral virtual machine. A virtual machine is defined as a set of parts and services (communications, scheduling, naming, etc.), which are defined independently of any specific platform and which are realized in platform-specific ways on different platforms. A virtual machine is a platform, and such a model is specific to that platform. But that model is platform independent with respect to the class of different platforms on which that virtual machine has been implemented.

This is because such models are unaffected by the underlying platform and, hence, fully conform to the criterion of platform independence.

2.4.3 Platform Specific Model (PSM)

A *platform specific model* is a view of a system from the platform specific viewpoint. A PSM combines the specifications in the PIM with the details that specify how that system uses a particular type of platform.

2.4.4 Model Transformation

Model transformation is the process of converting one model to another model of the same system.

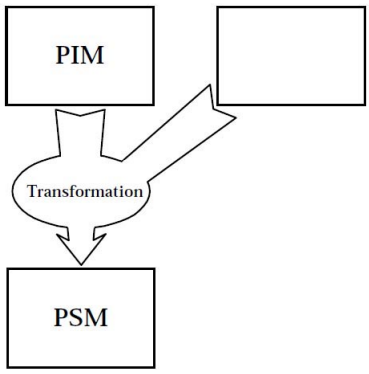

Figure 2.10: Model transformation. From PIM to PSM.

As shown in Figure 2.10, the *platform independent model* and other information are combined by the transformation to produce a *platform specific model* and there are many ways in which such a transformation may be done.

2.4.5 Applying *Model Driven Architecture* to Testing

As expressed by Dai [Dai04], the MDA abstraction levels can also be applied to test modeling.

> As shown in Figure 2.11, platform independent system design models (PIM) can be transformed into platform specific system design models (PSM). While PIMs focus on describing the pure functioning of a system independently from potential platforms that may be used to realize and

execute the system, the relating PSMs contain a lot of information on the underlying platform. In another transformation step, system code may be derived from the PSM. Certainly, the completeness of the code depends on the completeness of the system design model.

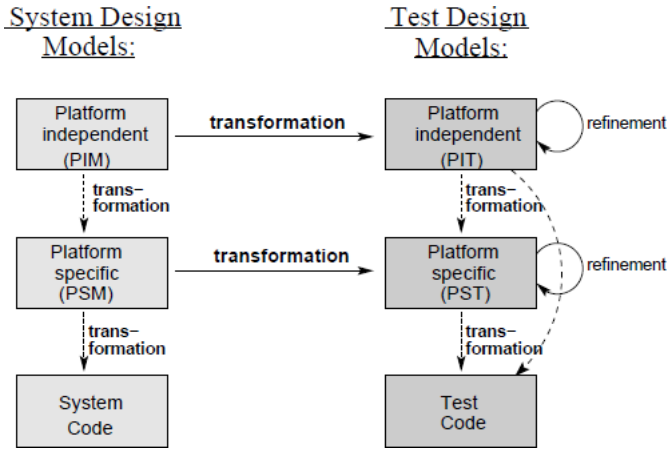

Figure 2.11: System Design Models vs. Test Design Models [Dai04].

The same abstraction in terms of platform independent, platform specific modelling and system code generation can be applied to test design models. Furthermore, test design models might be transformed from system design models directly. This enables the early integration of test development into the overall development process. Once the system design model is defined at PIM level, a platform independent test design model (PIT) can be derived. This model can be transformed either directly to test code or to a platform specific test design model (PST).

The same transformation technology can be used for deriving PSTs from the PSM. After each transformation step, the test design model can be refined and enriched with test specific properties. Although the transformed test design model may already contain static and dynamic aspects, the behavior has to be completed in order to cover unexpected system behavior as well. Also, test issues such as e.g. test control and deployment information has to be manually added to the test design model. At last, the test design model can be finally transformed into executable test code from either PST or PIT [Dai04].

2.5 The Eclipse Modeling Framework

The *EMF* is a modeling framework and code generation facility for building tools and other applications based on a structured *data model*.

The basic concept is that either Java classes, UML diagrams or also the XML schema's are done for describing the same kind of thing, that is a *data model*.

> EMF is a framework and code generation facility that lets you define a model in any of these forms, from which you can then generate the others and also the corresponding implementation classes. Figure 2.12 shows how EMF unifies the three important technologies: *Java*, *XML*, and *UML*. Regardless of which one is used to define it, an EMF model is the common high-level representation that *glues* them all together [BBM03].

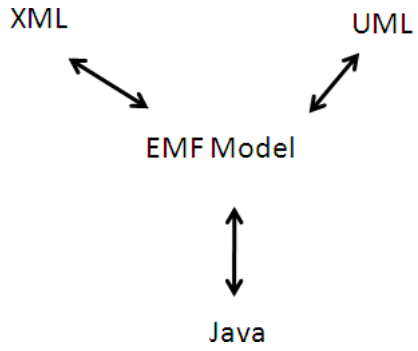

Figure 2.12: *EMF* unifies *Java*, *XML*, and *UML*.

The ecore metamodel

The model used to represent models in *EMF* is called *Ecore*. *Ecore* is itself an EMF model, and thus is its own metamodel.

For example, the model shown in the Figure 5.10 corresponds to an *Ecore* model. It can be easily noticed that the *Ecore* model corresponds almost to an UML model. In fact, the *Ecore* model corresponds to a subset of the *UML* modeling language.

XMI Serialization

XMI is a standard for serializing metadata. Thus, its used by the EMF to serialize *Ecore* metamodels (containing metadata).

Every *UML* tool has its own persistent model format. An *Ecore XMI file* is a *Standard XML* serialization of the exact metadata that EMF uses.

An example of this serialized file using this concepts is shown in the Appendix 9.2.

EMF summary

To summarize, the *EMF* constituting concepts are:

1. Ecore, and its XMI serialization, is the center of the EMF world.
2. A core model can be created from any of at least three sources: a UML model, an XML Schema, or annotated Java interfaces.
3. Java implementation code and, optionally, other forms of the model can be generated from a core model.

Chapter 3

State of the art

Several studies have been made regarding test case generation. These studies include interesting subjects such as the automatic test generation based on different models (specially those coming from UML), coverage criteria, data synthesis for evaluating the systems under test and model-driven architecture based test generation. In this section, the most relevant proposals will be exposed in order to explore the *state of the art* in this subject.

To begin with, the book of Robert V. Binder [Bin00] is one of the first and most complete referrals about designing and automating test suites for object-oriented systems. It explains why testing must be model-based and provides in-depth coverage of techniques to develop testable models (from state machines, combinational logic, and UML). It also presents several patterns that explain how to design test suites, how to tailor integration and regression testing for object-oriented code, how to test reusable components and frameworks, and how to develop test suites from use cases. Finally, it provides the *how-to* for application-specific test automation. This book served as a basis for several new studies that will be mentioned in the sections below.

3.1 Exploring execution paths

The references grouped in this section are those related with the examination of all possible execution paths that a business process can take. Although there is no documentation related directly to BPM diagrams used to derive and generate test cases, the papers exposed focus on other similar flow chart diagrams.

To begin with, a great many problems are naturally formulated in terms of objects and connections between them, or in other words, they can be reduced as the concept of *Graphs*.

The *Graph Theory* is a major branch of combinatorial mathematics and has been intensively studied for hundreds of years. The more known algorithms include the

exploration of the graph the find the shortest path from one node to the other, the possibility to search for a node inside the graph, among others.

The *Graph Theory* algorithms we are interested in are those related with the graph exploration to find all the possible paths between two nodes.

R. Sedgewick [Sed83] has written many books about the basic methods for solving problems which are suited for computer implementation, more known as *algorithms*. As graphs are an extended way of representing interconnected things, he does not forget of exposing the basic graph exploration algorithms.

One of the most famous algorithms to explore a graph is called *Depth-First Search*. This algorithm is a natural way to visit every node and check every edge in the graph systematically.

As this algorithm is unable to solve straightforwardly the paths retrieval problem, R. Simões [Sim09] has proposed all algorithm to retrieve cycles and paths in all kind of graphs. The proposed algorithm can be easily implemented and is highly modular; with minor changes it can be adapted to obtain different parameters from the graphs.

3.1.1 Model Based Approaches

Since the appearance of UML [OMG09a], models are being used for a big variety of purposes: communication, documentation, capturing details of the architecture and design of a system, among others.

Day by day, these documents became a high level language used to derive automatically lower level code. As a consequence, models are also used to generate tests which aim is the comparison of the actual system behavior with the expected one.

Many authors have considered these facts and studied it further to do a better test case generation. The researches done can give us a glimpse of the spectrum of models that are being exploited to generate test cases. Thus, from each one of them it can be extracted more than one valuable idea that can help to come upon with the most effective solution.

The previous studies are exposed below ordered by the model chosen by the respective authors. (Note that these models sometimes are coupled and used one together with the other).

State diagrams

State diagrams depict the various states that an object may have during its life and the transitions between those states.

Y.G. Kim *et al* [KHBC99] discusses the identification of control flow by transforming UML state diagrams into finite state machines. The control flow in UML state diagrams is identified in terms of the paths in an Extended Finite State Machines (EFSMs).

In general, there are infinitely many paths in EFSMs and hence is impossible to cover all these paths. For being able to restrict these infinite paths, they have explored three possible criteria: *path coverage*, *state coverage* or *transition coverage*. Test cases satisfying these criteria can be constructed in terms of *simple breadth* or *depth first searches*[Sed83] over EFSMs.

Likewise, J. Offutt *et al* [OA99], presents a technique to transform existing UML state diagrams into test cases by means of different concept for coverage criteria (*Transition coverage*, *Full predicate coverage*, *Transition-pair coverage* and *Complete sequence*). For these specified criteria, its possible to automate all the steps for generating test data. He does it by developing a test data generation tool named *UMLTest* that works integrated with Rational Rose[1]. Besides, the paper exposes the algorithms used by that tool to derive the test cases from a given UML state diagram.

These studies have shown the importance of defining a coverage criterion to be able to define how good the tests are. Not only the definition of the criterion is important but also the existence of several algorithms to exploit the different states and its relationships to generate the desired tests.

Summarizing, is important to remark the interesting issues addressed when generating test cases starting from state diagrams:

- The description of different coverage criteria ([OA99] and [KHBC99]).
- The use of an intermediate model to simplify the automatic generation of test cases (Extended finite state machines [KHBC99]) .
- The use of different algorithms to exploit the state diagrams ([OA99] and the *simple depth first search* [Sed83]).

Use cases

Use cases describe how external actors interact with the software system we are interested in creating. During this interaction an actor generates events to a system, usually requesting some operation in response [Lar02].

The UML use cases are the first good source for derivation of the software testing requirements, because they represent the high level functionalities provided to the end-user by the system[BLL04].

L. Briand *et al* [BL02] proposes a new system test methodology based on the artifacts produced at the end of the analysis development stage including use case descriptions and diagrams.

Use cases are used together with other modeling elements such as *activity diagrams* to express the sequential constraints; *interaction diagrams* along with the

[1]Rational Rose is an UML software design tool intended for visual modeling and component construction of enterprise-level software applications.

Object Constraint Language (OCL) specifying additional object interaction; and a *data dictionary* to provide an addition to an informal description of classes, attributes and operations (including pre-post conditions) and invariants in OCL.

Based on the previous paper, C. Nebut *et al* [NFTJ03] [NFTmJ03], proposes a new simple contract language for functional requirements to parametrize use cases.

The ordering of use cases is expressed with a light formalism, such that the use case diagram remains a good communication basis with domain experts. That is why the use of preconditions and postconditions is a good way to easily and quickly express the mutual obligations and benefits among functional requirements, expressed with use cases.

When dealing with the automatic test generation, they build a use case transition system and then exploit it with several criteria to generate relevant test objectives. Based on the different concepts of coverage criteria, they also exposes the different algorithms needed to generate the *Use Case Transition System (UCTS)* representing the valid sequences of use cases.

Finally, J. Hartmann *et al* [HVFR05], makes mention to the complexity of managing use cases when considering complex use cases and test scenarios, which include multiple, nested alternative flows. That's why he proposes the conversion of the existing documents into activity diagrams.

Using use cases as a way of generating test cases can be a good idea taking into account that many systems nowadays are specified by them. But, on the other side, it can be bothersome to do an extra-work to make all the object relationship fully consistent.

From this we conclude some aspects to be taken into account regarding to the use cases approaches:

- The necessity of expressing how the objects interact with each other. (Addressed by the use of pre-post conditions as well as the OCL [BL02], [NFTJ03], [NFTmJ03])
- The need of another model to represent the transitions among the different use cases (Use of activity diagrams [BL02])
- It can become unmanageable when considering complex use cases and test scenarios ([HVFR05]).

Sequence diagrams

Sequence diagrams are another useful way to describe the behavior of some portion of the system. They specify precisely the set of objects and the sequences of message exchanges that are involved in various scenarios.

B. Rumpe in [Rum03] mentions the sequence diagrams as a model that gives the advantage of being able to describe the method calls as well as the desired interactions that can be checked during the test execution. The sequence diagrams, in this context, are used together with the *Object Constraint Language (OCL)*).

The most recent approach was suggested by A. Nayak *et al* [NS10]. They use sequence diagrams to represent the behavior of the system under development. They transform this sequence diagram into an intermediate form (the *structured composite graph*) which unambiguously and in a structured way shows all the interaction operands and flow of control of these operands. In this paper the algorithm to transform a sequence diagram into a SCG (structured composite graph) is presented (As well as a way to derive test data that will be exposed in Section 3.2.

To summarize its important to remark these facts when considering sequence diagrams to generate test cases:

- The sequence diagrams are very close to the specification of system calls ([Rum03]).
- The use of OCL to express interaction between objects ([Rum03], [NS10]).
- The use of an intermediate model to facilitate the test case generation (The *structured composite graph* [NS10])

Activity diagrams

An activity diagram offers rich notation to show a sequence of activities. It may be applied to any purpose (such as visualizing the steps of a computer algorithm), but is considered especially useful for visualizing business workflows and processes, or use cases [Lar02].

As this diagram is the most similar notation to the one that will be considered in this thesis (the business process modeling notation or BPMN), all the findings in this area will be of relevant importance.

The first solution using activity diagrams was the one of L. Briand *et al* [BL02] (Section 3.1.1) used together with the already mentioned concept of use cases.

Every activity diagram specifies an infinite number of paths (given by existing loops). For this reason they use a similar strategy to the one used to test loops in code (i.e., making sure that each loop is bypassed - if possible, take only once, a representative or average number above 1, and maximum number of times). Paths are first determined through a *simple depth-first search* [Sed83].

The next step is to determine dependencies in terms of actual parameter values between the use cases in a path. Dependencies are needed in order to identify data flow between use case executions, something that will be necessary for the generation of test input data. They can simply be determined based on the actual parameters in the activity diagram, such parameters serving as placeholders for actual values.

Besides, the construction of use case sequences may need to instantiate several times, with different values. For this purpose, the tester should choose the scale of the test to take place and the constraints on dependency loops.

J. Hartmann *et al* [HVFR05], mention the activity diagrams as a very well suited tool to reflect the user input and system-response paradigm by being partitioned into swim-lanes. They propose the annotating of the activity diagrams to provide additional information that can be useful when generating the test cases.

For test generation they use a product named *TDE* that processes a test design written in the test-specification language or TSL. This language is based on the category-partition method, which identifies behavioral equivalence classes within the structure of a system under test. A TSL test design is created from the activity diagram by mapping its activities and transitions to TSL partitions and choices.

The generation procedure consists in a recursive, directed graph built by the TDE, having a root category/partition and containing all the different paths of choices to plain data choices. This graph may contain cycles depending on the choice definitions and is equivalent to the graph of the global state machine. A test case is one instance of the initial data category or partition, that is, one possible path from the root to the leaf of the reachability tree for the graph. The content of the test case consist of all data values associated with the edges along a path in the paragraph.

X. Bai's *et al* [BLL04] recalls the definition of *thin threads* as an usage scenario in a software system from the end user's point of view. From this basis, they propose the generation of test scenarios deriving the thin thread in the form of inter-related *thin thread trees*, *condition trees* and *data-object trees* obtained from activity diagrams. Their approach requires manual conversion of an activity diagram into an activity *hypergraph*[2]

L. Wang *et al* [LJX+04] introduce a gray-box approach [3] and tool for generating test cases from activity diagrams. Their approach, like the previous one, could not be fully automated. Another restriction is placed in the activity diagrams structure, being able to contain only a single initial node and a single final node (Going against the UML2 specification which states that an activity can have multiple initial nodes). Besides, their approach restricts to a maximum of two-threads that have only sequential call actions.

Interestingly, L. Wang *et al*'s algorithm does not appear to differentiate between branch nodes and fork nodes. Even with the *two-thread restriction* its not possible to

[2]A hypergraph is a graph in which generalized edges (called hyperedges) may connect more than two nodes.

[3]Gray box method combines the white box method and the black box method. It extends the logical coverage criteria of white box method and finds all the possible paths from the design model which describes the expected behavior of an operation. Then it generates test cases which can satisfy the path conditions by black box method. It can find problems which used to be ignored by both black and white method [LJX+04].

capture a pair of threads in a single scenario. This means that their approach may exclude many scenarios from a model, while at the same time producing incomplete traces.

D. Xu *et al* [XLL05] present an approach that employs adaptive agents [4] to extract thin-threads from activity diagrams. They list the set of test-cases derived from the artifacts between an activity diagram's initial and final nodes in an output log file. These set of test cases can be clustered together to form a path through the activity diagram's artifacts. From these clustered paths it can be established which usage scenarios have been captured and which have not.

Furthermore, they introduce four categories to describe specific, simple situations. These are: (1) Atomic Fork Join; Simple Fork Join (SFJ); (2) Simple Nested Fork Join (SNFJ); and (3) Branch Nested Fork Join (BNFJ).

Based on the studies mentioned above ([BLL04], [LJX+04] and [XLL05]), R. Chandler *et al* [CLLW05] discusses an approach to generate usage scenarios from activity diagrams that does not involve manual pre-processing.

In this approach they use *thin threads trees* as a foundation for test-case generation, coverage analysis and test-result analysis according to [oD00].

They also augment the list provided by D.Xu *et al* mentioned above, adding: (4) Basic Fork Join (BFJ); (5) Looping Fork Join (LFJ) ; (6) Skipping Fork Join (SFJ); (7) Nested Fork Join (NFJ); (8) Skipping Nested Fork Join (SNFJ); and (9) Looping Nested Fork Join (LNFJ).

The analysis of R. Chandler *et al* includes the comparison between the three solutions exposed before. The comparison is done in terms of *path coverage*, *condition/branch coverage*; and *data coverage*. Bai's *et al* [BLL04] manual approach accurately produces all scenarios; while Wang's *et al* [LJX+04] *gray-box* method treat each trace as a separate scenario and Xu's *et al* [XLL05] although producing many of the usage scenarios, fails when looping is included in the trace.

A new interesting approach is given by C. Mingsong *et al* [MXX06]. They begin proposing the generation of abundant test cases for a Java program under testing starting from an activity diagram. Then, by running the program with the generated test cases, they manage to get the corresponding execution traces. Last, by comparing these traces with the given activity diagram according to the specific coverage criteria (*activity coverage*, *transition coverage*, *simple path coverage*), they get a reduced test case set which meets the test adequacy criteria.

In the study of C. Mingsong *et al*, they apply the *program instrumentation* technique. This technique's main idea is to insert some probes which are one or more test statements inserted in the original source code for recording dynamic informa-

[4]The adaptive agents proposed in the paper were conceived to copy the behavior of the Maxococcus xanthus bacteria. The objective of the agents exploration is to build the test scenarios from the corresponding activity diagram.

tion. When executing the instrumented program, the probes are executed and the executing behavior data is thrown out and can be recorder.

Similarly, to avoid the test case explosion, H. Kim *et al* [KKBK07], build an *I/O explicit activity diagram* from an ordinary activity diagram and then transforms it to a directed graph. From this graph, test cases for the initial activity diagram are derived. This conversion is performed based on the single stimulus principle, which helps to avoid the state explosion problem in test generation for a concurrent system.

The most recent research is the one given by A. Nayak *et al* on [NS09]. Their approach take as main premise the complexity of interpreting activity diagrams directly. Therefore, to avoid ambiguous interpretations they provide automated methods for analyzing complex dependencies that arise within nested structures. They propose the identification of each construct inside the diagram and the mapping to a well structured model before generating the test cases. These well structured models are called *intermediate testable model (ITM)* and they are used to generate test scenarios.

Activity Diagrams seems one of the best ways to represent the system behavior. This is reflected in the number of studies done using them as a basis for Test Case generation. This is because they are simple to understand and efficient enough to model most of the real business process situations.

From all the exposed papers, its important to summarize:

- The need of identifying constructs inside the diagram such as forks, joins or loops ([XLL05], [CLLW05]).
- The need of generating an intermediate model to exploit all the possible combination of paths (Thin thread trees [CLLW05]; condition trees, data-object trees and hypergraphs [BLL04]; and an Intermediate Testable Model [NS09]).
- The possibility of annotating activity diagrams to provide additional information when generating the final test cases.
- The existence or invention of different algorithms to explore the activity diagrams. (the *simple depth first search* [Sed83], *anti-ant like agents* [XLL05], the *gray-box* approach [LJX$^+$04], the *program instrumentation* technique to extract traces and its comparison with the activity diagrams, used by [MXX06]).
- The presence of a tester role to specify further details when generating the test cases ([BL02])
- The existence of a language to describe tests called TSL ([HVFR05])
- The need to restrict the test case generation by defining a coverage criterion ([CLLW05], [MXX06], [KKBK07], etc).
- The need to avoid the explosion of test cases ([MXX06], [KKBK07]).

3.2 Test data generation

Other important aspect of test case generation is the arbitrary data needed to test the system. The papers grouped in this section explore how to provide automatic data input needed for executing the generated test cases.

The most recent approach to test data generation was suggested by A. Nayak *et al* [NS10]. On their research they expose in detail the current methods for test data generation:

> Test data generation methods ([Kor90], [Kor96], [MMS01], [Cla76], [How77], [Off91], [FK96]) have been widely applied to unit testing and module testing in function oriented programming. The methods employ structural testing criteria to extract required information from programs.
>
> Depending on whether the control or data flow aspects of an implementation is chosen, the structural testing criteria can be classified as *control flow* and *data flow* based criteria.

- **Control flow based criteria:** Analyze the control flow such as statement, branch, loop and concurrent constructs of an implementation ([Bei03], [How75], [Nta88], [Tay83]). For this purpose, the control flow of a program is usually represented by a *control flow graph*, where the nodes are either a decision node or a node representing single entry and single exit sequence of statements known as block node. The edges represent possible control flows between nodes.

- **Data flow based criteria:** Require the data flow associations between definitions and uses of variables to be exercised [RW85]. For a given test criterion, the test data generation methods find a test input to exercise various test requirements. For example, *statement coverage* finds a test input for each program statement, *branch coverage* finds an input such that the execution traverses a specified edge of the control flow graph associated to the program and path coverage finds the input causing the execution of a specified path.

In this context, they have classified the different proposed methods for automatic test data generation in three categories.

- **Random test data generation:**Generates test input values using a random number generator. ([Kor96], [MMS01]).

- **Path oriented test data generation:** Identify a set of paths and then generate input to execute the selected paths. For these methods, test generation is based on:

  - *Symbolic execution:* This method assign symbolic values to variables. For example, an input variable x is assigned a symbolic value such as x = x0. After a few executions down the path, the value of the variable becomes a complex expression. ([Cla76], [How77], [Off91])
  - *Dynamic execution:* This method is based on actual execution of a program under test and test data is developed using actual values of input variables. In the beginning, all input variables are set to take some initial values. When the program is executed with this data, the flow is monitored which helps in determining whether the test requirements are satisfied or not. ([Kor90])

- **Goal oriented test data generation:** For the selected node in a program known as goal node *g*, the procedure finds program input so that the node *g* will be executed irrespective of the path taken. ([FK96])

3.2.1 Model Based Approaches

As said before, models are being used for a big variety of purposes. In this case, models are also exploited to extract test data. The different approaches regarding each model are enumerated below.

Sequence Diagrams

The method explored by Pilskalns *et al* [PAK+07] merges information coming both from sequence diagrams and UML structural views. This information is used to build instances of a class. The test data generation uses Binder's domain analysis approach [Bin00] for identifying the set of variables occurring in conditional expressions. Using these variables, they create partitions and select the test values.

Another approach is the one done by Dinh-Trong *et al* [DtGF06]. Their study introduces a representation called the *Variable Assignment Graph (VAG)* that integrates relevant information from class and sequence diagrams. The VAG is used to derive test input constraints that are then solved by the constraint solver, Alloy [JSS00]. The novelty of this approach is that it considers both constraints on variables with primitive datatypes (as done in other test generation approaches), and constraints on object configurations characterized by the class diagram view of a system.

As said previously in the Section 3.1.1, the approach developed by Nayak [NS10] starts interpreting *sequence diagrams* as a *structured composite graph* and after that they proceed with the data synthesis.

The data synthesis phase consists in finding the test input that satisfies all the constraints along the path. The proposed synthesis procedure steps are:

- Derivation of the constraints for the specified scenario
- Solving the constraints along the scenario
- Generating test data for finding test input to the variables involved in the scenario.

Communication Diagrams

Samuel *et al* [SMK07] selects test cases (message paths inside the tree representation of an UML communication diagram. The test data is then solved by the function minimization technique proposed by Korel [Kor90] that is applied on predicates selected from the message paths. The main drawback is that the test data for different types of data is not addressed and integer data type is considered as a default data type. This may result into solution with need of some other technique to complement it.

State Diagrams

The approach of Weileder *et al* [WS08], uses state machines, class diagrams and OCL expressions to generate test cases. Classification of the variables in OCL expressions is provided by recognizing the variables that can alter the value of attributes from those that cannot alter them. When generating the test cases, the variables are identified and the creation of partitions of the value ranges for input variables is done. The test cases are then created b selecting representative value for each parameterized event of the path.

Activity Diagrams

J. Hartmann *et al* [HVFR05] addresses the problem of data generation of system tests from UML activity diagrams. The idea of their solution is to assign a category to all the variables in the diagram. Besides, based on the test requirements, equivalent classes to those in the system under test are created. The existing problem with this approach is that structures, such as concurrency loops, and their nested combinations are not addressed. In addition, simple predicates are evaluated as the branch predicates.

Chapter 4

Application Context

This chapter exposes the context in which this work is supposed to be applied as a first instance. However, the idea is to provide a generic approach that can be applied to different contexts, independently of the chosen platforms. Therefore, this chapter can give a broad idea of a possible application context for the proposed work.

4.1 The Testing Framework

The framework that is used as a starting point for this work is the one mentioned by Fraternali *et al* [FT10]. This framework make use of the previously shown MDA concepts explained before in Section 2.4.

The proposed framework is described as follows:

> Figure 4.1 shows an overview of the models involved in our framework. For each one of the MDA abstraction levels, we define both a metamodel of the Web application and a metamodel of the test case:
>
> - at the CIM level, the modeling language is BPMN and the Computation Independent Tests (CITs) are modeled in our BPMN-Test metamodel;
> - at the PIM level, the modeling language is WebML and the Platform Independent Tests (PITs) are modeled in our WebML-Test metamodel;
> - at the PSM/code level, Platform Specific Tests (PSTs) are Web navigations represented as scripts of a Web testing suite (e.g. the Canoo WebTest system2).

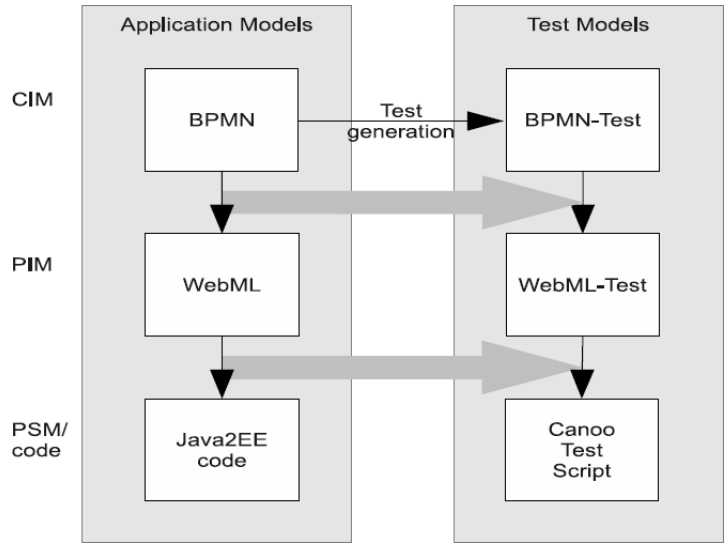

Figure 4.1: Overview of the transformation framework.

In the design of the test case metamodels we seek maximum simplicity and extensibility. The metamodels are based on a common core. They comprise a container class *TestSuite* that can be decorated with information about the application configuration.

TestSuite contains multiple *Tests* composed by ordered sets of *Steps*. Each *Step* specifies the identifier of an application session, e.g. useful for distinguishing actions performed by concurrent users of the system.

Step is specialized in two abstract classes that have to be subclassed for each concrete test case metamodel: an *ActionStep* activates some elements of the application model, referenced by an identifier, while an *Assertion-Step* represents the evaluation of a predicate over the application state. Given an application domain, new *ActionStep* or *AssertionStep* subclasses can always be defined for domain-specific tests.

Excerpts from the BPMN-Test is shown in the Figure 4.2. It is easy to identify a reference to the specific concepts of the respective application models.

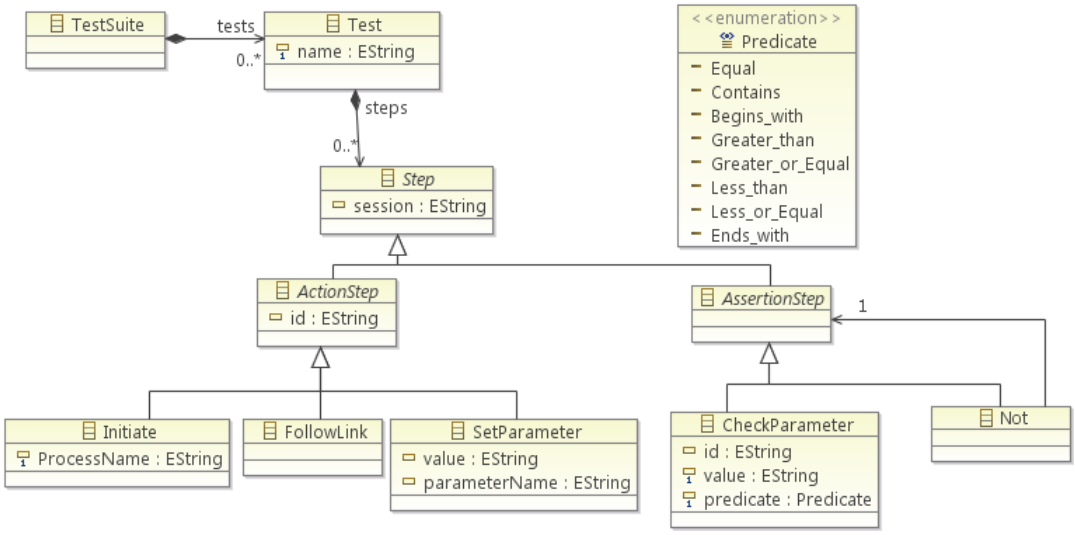

Figure 4.2: BPMNTest Metamodel

A *BPMN-Test* model allows one to initiate a process instance, follow its links and insert values in the process instance variables. The only assertion provided checks the value of a process instance variable, but new assertions can be introduced by defining new subclasses. Finally a *Not* assertion allows one to negate the truth value of a referenced assertion.

The described *BPMN test scenario* consists on a single *Test* with this sequence of *Steps* elements, for example:

```
1a. initiate (session="1", id="lane1")
2a. setParameter (session="1", id="title", value="Car Rental")
2b. setParameter (session="1", id="expense", value="50")
3c. followLink (session="1", id="link2")
3a. initiate (session="2", id="lane3")
3b. checkParameter (session="2", id="title", value="Car Rental", predicate="equal")
3c. checkParameter (session="2", id="expense", value="50", predicate="equal")
4a. setParameter (session="2", id="Receipts status", value="true")
4b. followLink (session="2", id="link4")
4c. followLink (session="2", id="link5")
```

4.2 WebRatio

The auto-generating tool *WebRatio BPM* was chosen as the business processes modeling tool that will provide the starting point models and required files to test the proposed framework.

WebRatio BPM [BBF10] is an Eclipse-based tool that allows the production of applications dedicated to the *Business Process Management*. The innovation and the competitive edge are:

- The automatic generation of a BPM application starting from its BPMN model. Many other tool allow to model a BPMN diagram and obtain a running Web application, but none of them has a fully-automated generation. WebRatio BPM will be able to automatically generate all the feature of the Web application: the logic layer, the data layer and the presentation layer
- The extreme prototyping approach. Due to the fully-automated generation, WebRatio BPM will allow the generation of an unlimited number of prototypes before getting the final solution. This will help developers in keeping their development process much closer to their customers needs, through frequent milestones and check points
- No runtime server. Unlike any other BPMS tool, the application generated by WebRatio BPM will be uniquely based on the Java Servlet standard, without the need of any other server or license

4.2.1 Business Process Model

Based on the Business Process Modeling Notation provided by the OMG (Object Management Group) [OMG09b], *WebRatio* offers their graphical tool to represent Business Processes. The tool contains the main modeling elements (explained in Section 2.2) needed to depict a complete business process. This elements are summarized below.

Flow Objects

Flow objects are the main describing elements within BPMN, and consist of three core elements (*Events*, *Activities*, and *Gateways*):

1. Events

In *WebRatio* an *Event* is represented with a circle and denotes something that happens in a given moment. The icons shown inside the circle denote the type of event (e.g. circle with an envelope denotes a message, the one containing a clock denotes time).

Events are also classified as *catching* event if they catch an incoming message to start the process or *throwing* event if they throw a message at the end of the process.

Figure 4.3: Start events provided by *WebRatio*.

Figure 4.4: End events provided by *WebRatio*.

- **Start event:** Acts as a trigger for the process; In *WebRatio* is indicated by a single narrow border surrounding the circle. This kind of event can only be of *catching* type and as a consequence is shown with an open (outline) icon.
- **End event:** Represents the result of a process. It is indicated by a single thick border surrounding the circle. This type of event and can only of *throwing* type. As a consequence, they are shown with a solid icon.
- **Intermediate event:** Represents something that happens between the start and end events. This event is indicated by a tramline border. It can be both *throwing* or *catching* types (using solid or open icons as appropriate). For example, a task could flow to an event that throws a message across to another pool and a subsequent event waits to catch the response before continuing.

2. Activities

An *Activity* is represented in *WebRatio* as a rounded-corner rectangle that describes the kind of work that must be done.

Figure 4.5: Activities provided by *WebRatio*.

- **Task:** A Task is an atomic activity that is included within a Process. A Task is used when the work in the Process is not broken down to a finer level of Process Model detail.

- **Sub-process:** Is a compound activity in that it has detail that is defined as a flow of other activities. A Sub-Process is a graphical object within a Process Flow, but it also can be *opened up* to show another process (either *embedded* or *reusable*).

- **Transaction:** A transaction is another form of a sub-process in which all contained activities must be treated as a whole, i.e., they must all be completed to meet an objective, and if any one of them fails they must all be compensated (undone). *Transactions* are differentiated from *expanded sub-processes* by being surrounded by a tramline border.

3. Gateway

In *WebRatio* there are different kind of *Gateways* that can be combined together to model different behaviors within a flow of the business process.

Figure 4.6: Gateways provided by *WebRatio*.

- **Exclusive Gateway:** Locations within a business process where the flow can take two or more alternative paths. For a given instance of the process, only one of the paths can be taken. They are represented in *WebRatio* as a diamond with an X in the middle.

- **Inclusive Gateway:** Branching point where the alternatives are based on conditional expressions contained within outgoing Sequence Flow. However, in this case, the *True* evaluation of one condition expression does not exclude the evaluation of other condition expressions. In *WebRatio* they are represented as a diamond with a circle inside.

- **Parallel Gateway:** They provide a mechanism to synchronize parallel flow and to create parallel flow. These Gateways are not required to create parallel flow, but they can be used to clarify the behavior of complex situations where a string of gateways are used and parallel flow is required. They are represented as a diamond with a plas symbol in the middle.

- **Complex Gateway:** BPMN includes a Complex Gateway to handle situations that are not easily handled through the other types of Gateways. Complex Gateways can also be used to combine a set of linked simple Gateways into a single, more compact situation. They are shown in *WebRatio* as a diamond with a star inside.

- **Exclusive Event Gateway:** It refers to a branching point where alternatives are based on an event that occurs at that point in the process. They are shown in *WebRatio* as a diamond with a pentagon figure inside.

In *WebRatio*, the gateways execution can be configured as follows:

- **User:** A typical workflow gateway, where a human performer makes the choice with the assistance of a software application.
- **Service:** A gateway that uses some sort of service, which could be a *Web service* or an *automated application*. The service gateway can be also ruled by a condition. There are two ways to control the gateway by a condition:
  - *Expression:* The expression can be expressed like a *logical condition* using the parameters linked to the gateway (For example, a condition may be written as: $Age \geq 20$ where age is a parameter and 20 is the specified value for the condition). On the other hand, it can be expressed by means of a script that can evaluate a more complex condition.
  - *Literal:* The literal condition assumes different literal values. Each value will be assigned to a different subsequent flow arrow. An example is shown in the Figure 7.12 where the system checks if the value is *Approved* or *Not Approved*. Depending on those values, the flow will continue a different flow (*Inform Reject Reason* or *Make Administrative Task*).

Connecting objects

They consist of three types: Sequences, Messages, and Associations.

- **Sequence Flow:** Is represented with a *solid line* with arrowhead and depicts in which order the activities will be performed. The sequence flow may also have a symbol on its start point: a small *diamond* on top of it indicates one of a number of conditional flows from an activity whereas a *diagonal slash* indicates the default flow to be taken.
- **Message Flow:** Is represented with a *dashed line*, an open circle at the start, and an open arrowhead at the end. It tells us what messages flow across organizational boundaries (i.e., between pools). A message flow can never be used to connect activities or events within the same pool.
- **Association:** Is represented with a *dotted line*. It is used to associate an *Artifact* or a text to a *Flow Object*. It can indicate some directionality using an open arrowhead (toward the artifact to represent a result, from the artifact to represent an input, and both to indicate it has been read and updated).

No directionality would be used when the *Artifact* or text is associated with a sequence or message flow.

Swim Lanes

Swim lanes are a visual mechanism of organizing and categorizing activities, based on cross functional flowcharting.

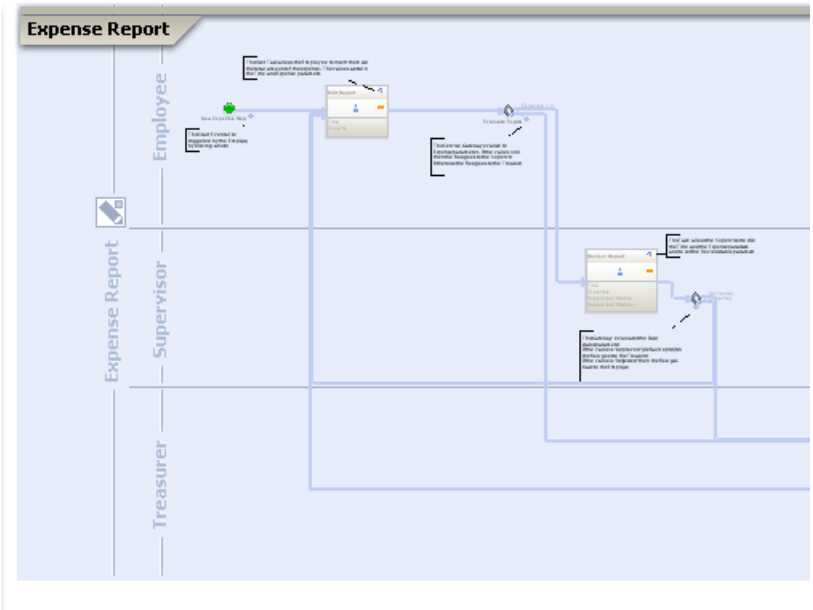

Figure 4.7: Swim lanes in *WebRatio*. Each lane correspond to a different role (Treasurer, Supervisor and Employee).

The swim lanes are represented in *WebRatio* as follows.

- **Pool:** Represents major participants in a process, typically separating different organizations. A pool contains one or more lanes. A pool can be open (i.e., showing internal detail) when it is depicted as a large rectangle showing one or more lanes, or collapsed (i.e., hiding internal detail) when it is depicted as an empty rectangle stretching the width or height of the diagram.
- **Business Object:** A Business Object represents an entity in the business domain that the processes are designed to support. Its composed by a set of properties. A property is defined by a *name*, a *type* (e.g string, integer) and a set of *predefined values*.

- **Parameters:** A pool is characterized by a set of parameters. A parameter is an information managed by the process and can be of two types:
  - *Simple parameter:* Defined by a *name*, a *type* (e.g. string, integer), a selection policy (single or multiple) and a set of predefined values.
  - *Business object parameter:* A reference to a *Business Object*.
- **Lane:** A Lane is a sub-partition within a *Pool* and will extend the entire length of the Pool, either vertically or horizontally. Different Lanes can be placed within a single *Pool* and they represent the involved actors. These actors can be both humans, automatically managed or delegated to other systems. Lanes are used to organize and categorize activities (as *Task*, or *Subprocess*, or *Events*, or *Gateways*) within a Pool.

Artifacts

Artifacts allow developers to bring some additional information into the model so that it becomes more readable. There are three predefined Artifacts and they are:

- **Data Objects:** They show to the reader which data is required or produced within an activity.
- **Group:** Is represented with a *rounded-corner rectangle* and *dashed lines*. Its used to group different activities without affecting the flow in the diagram.
- **Annotation:** An Annotation is used to give additional information or details to facilitate the reader's understandability.

4.2.2 Data Model

Together with the business process model, if attached to a *Web Project*[1], *WebRatio* generates automatically a data structure as shown in the Figure 4.8.

[1] *Web Ratio* provides the facility of creating web applications starting from a BPM together with WebML models (WebML is a modelling language for describing web interacions). To create a Web Project starting from a BPM, it is necessary to attach a BPM project related with a Web Project. When this is done, the default Data Model generated automatically by *Web Ratio* is the one shown in the Figure 4.8

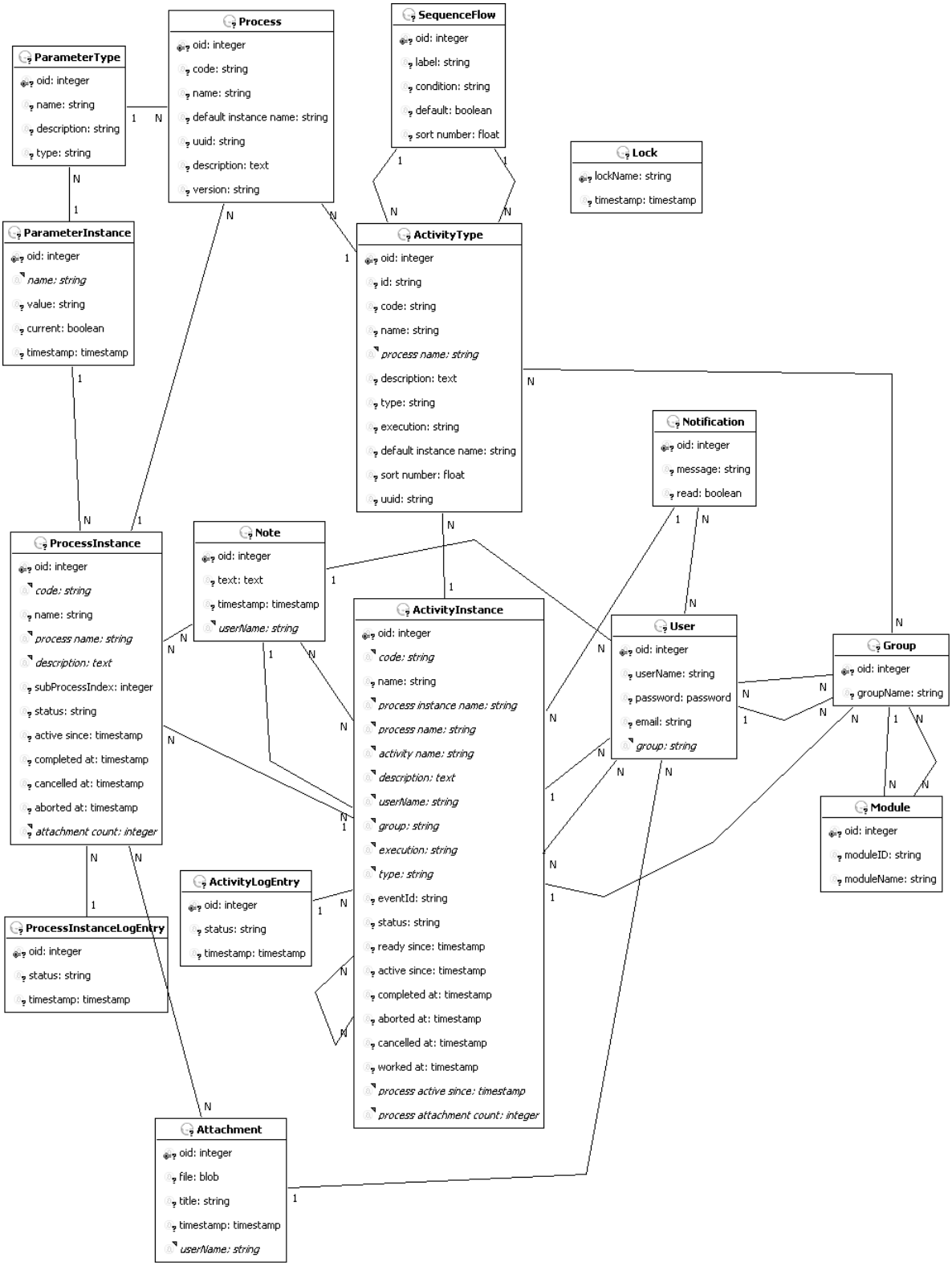

Figure 4.8: The default data model generated by *WebRatio* to contain the business process excecution.

A *Process* represents a whole BPMN diagram, and includes a list of *Activities*, each described by a set of input and output *ParameterTypes*. A *Case* is the instantiation of a process, and is related to the executed *Activity Instances*, with the respective actual Parameter Instances. The evolution of the status history is registered through *CaseLogEntry* and *ActivityLogEntry*. *Users* are the actors that perform a task and are clustered into *Groups*, representing their roles. The transformation from the extended BPMN to the Process Metadata is a relational encoding of the BPMN concepts: each process model is transformed to a *Process* instance; each activity is transformed into an *Activity* instance; each ow arrow is transformed into a *nextActivity/previousActivity* relationship instance; each guard condition is transformed into a *Condition* instance [BBF10].

Chapter 5

Generating Tests from BPM

In this chapter, the proposed approach for the generation of a *Test Suite* starting from a *Business Process Model* is presented.

5.1 Overview

From now on, the proposed methodology will be called ReTB (**Re**trieving **T**est Scenarios from **B**usiness Process Models).

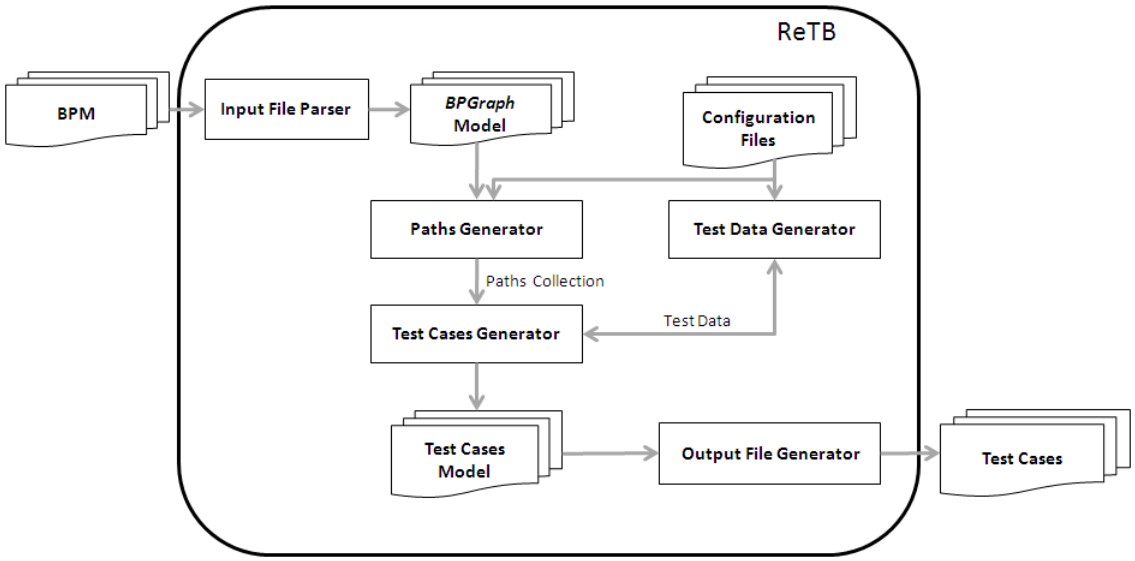

Figure 5.1: An overview of ReTB

In the first step, all the model elements present in the *Business Process Model*

(tasks, gateways, lanes, etc) are retrieved by an *Input File Parser* and transformed into a graph structure that will be named as *BPGraph* or *Business Process Graph*. The *BPGraph* will contain all the information of the *Business Process Model* but with a slightly different shape to the incoming one.

In the next step, the *Paths Generator* will transform the given graph into a set of execution paths. Each path will correspond to a feasible scenario of the business process.

Subsequently, the *Test Cases Generator* will take the collection of paths and will process each path individually. Each of them will be afterwards derived into one or more *Test Cases*. Each time a path is evaluated, the *Test Cases Generator* will call the *Test Data Generator* in order to retrieve different test data values that will be the ones associated to the different *Tasks* and *Gateways*. The result of this step is a *Test Cases Model* containing all the platform independent information needed to test the *Business Process Model* received as input.

Finally, this model will be taken by an *Output File Generator* that will be translated into a file that can be used later to continue, for example, with the platform dependent test cases execution.

An important remark about the proposed model is the possibility of interchanging the algorithms and ways of working of each of the constituting proposed component. This gives to the proposal a great advantage on flexibility for obtaining the expected results, with the possibility of implementing them in different environments, with different performances and different behavior, if needed.

5.2 Capturing the model element details

This section describes the first step of the proposed approach.

In this step, the *Business Process Model* is taken as an input and parsed into a known data structure that will be called *BPGraph* (Because it will contain useful information both from the *BPM* and the *Graph* structure).

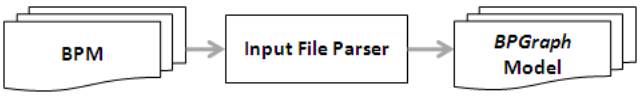

Figure 5.2: The first step of the *ReTB*: The *Business Process Model* is converted into a *BPGraph* Model by means of the *Input File Parser*

Depending on the type of tags, the different modeling elements e.g. tasks, transitions, lanes and gateways are extracted. For this purpose, an *Input File Parser* is considered. This parser stores the above mentioned information into a graphical

representation. This graph representation, in fact, depicts the connectivity between different nodes. The decision of having this additional model is not only to capture all the information required to generate the test cases but also being a suitable representation that enables the graph exploration as suggested in many existing algorithms.

5.2.1 The *BPGraph* Model

The chosen model is meant to be simple, understandable, extensible and as much generic as possible. This is shown in the Figure 5.3.

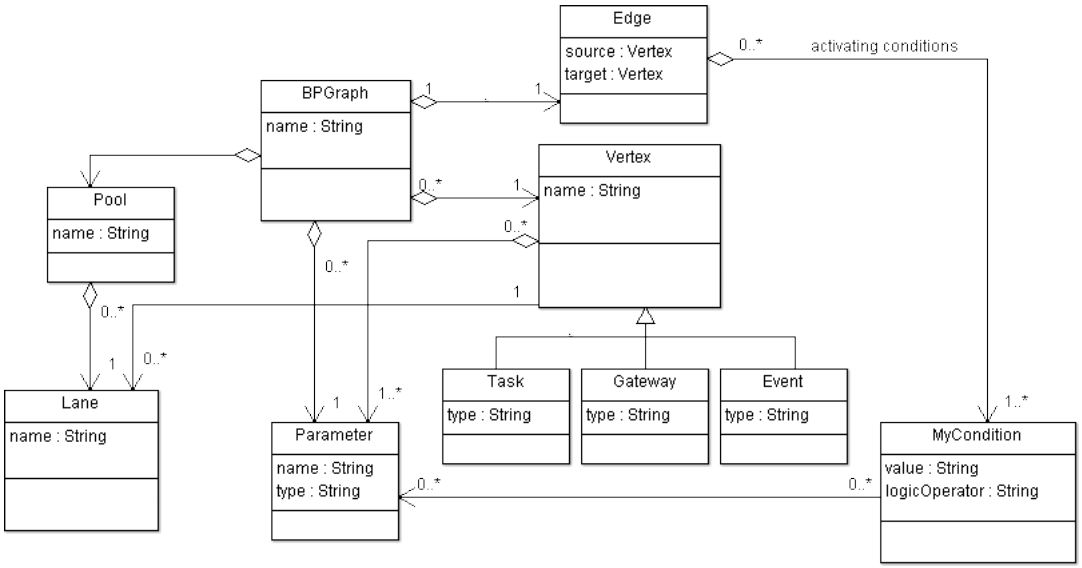

Figure 5.3: The *BPGraph* basic structure

Basically, the *BPGraph* contains all the information of the *Business Process Model* (e.g Tasks, Gateways, Events and Lanes) in the form of a *Graph* (Consisting in *Vertexes* and *Edges* connecting them). Thus, *Tasks*, *Gateways* or *Events* are considered *Vertexes* and the *Transitions* are the *Edges*.

5.2.2 Participants

The classes present in this model are:

- **BPGraph**: It represents the concept of the graph mixed with the one of the business process model. It contains a reference to all the elements inside the process. In other words, it contains:

  - The form expected by any directed *Graph*: A collection of *Vertexes* and a collection of *Edges* connecting the vertexes.
  - All the information captured from the *Business Process Model*: Pools, lanes, tasks, events, transitions and gateways.
  - Additional information about the business process: The *Parameters* that are seen, modified or evaluated in tasks or gateways and the *Conditions* activating the different edges.

- **Vertex**: Represents a node in the graph as well as tasks, gateways or events in a business process model. Therefore, the subclasses of *Vertex* are:
  - *Task*: Represents any kind of task inside the business process. It can be either a *Manual*, *User* or *Service* task.
  - *Gateway*: Represents any gateway of the business process. It can be either *Exclusive*, *Inclusive* or *Parallel*.
  - *Events*: Represents any event occurring in the process. It includes the concept of *Start* and *End* events.

- **Edge**: Is the directed connection between two *Vertexes*. The edge can have conditions that are required to activate the path between the source and target vertexes.

- **Parameter**: Represents a parameter being modified or read by any of the *Vertexes*.

- **Condition**: The condition represents an expression over a *Parameter*. Its formed by a *logic operator* (lower, lowerOrEquals, higher, higherOrEquals, equals, different), a *value* and the *parameter* that is evaluated in the condition. For example, a condition may be *Age minor than forty*, therefore, the *Parameter* is *Age*, the *Logic operator* is *lower* and the *Value* is *40*.

- **Pool**: Represents the same concept of pool of a *Business Process Model*.

- **Lane**: Represents the same concept of lane of a *Business Process Model*.

5.2.3 Consequences

The main benefit of using such a model is to make converge all the different *Business Process Model* into a simple and usable one. This is needed because, even if a standard is already defined, the reality is that each existing business modeler tool uses their own customization. In this way, the *BPGraph* model unifies the *Business Process Model* structure and a *Graph* basic structure (Collection of *Vertexes* and *Edges*).

On the other side, as there are many different ways to explore graphs, to model the business process in the *Graph* terms gives the benefit of doing any personal or adapted approach using graph exploration algorithms.

Besides, its important to remark that most of the *Business Process Models* does not contemplate the formal description of *Parameters* or *Conditions*. Even if this can be a good practice (to, for example, enable the complete or partial automation of the execution of the process in a chosen environment), the proposed graphical structure is supposed to work either if they are not expressed formally. In this case, as there will not be any *Condition* to enable the *Edge*, this will behave as a normal *Transition* between to vertexes. On the other side, the *Parameters* will not be evaluated if they are not defined for the process.

5.2.4 *Business Process Model* example

From this step onwards, an example of business process model is given in order to facilitate the explanation of the different chosen steps.

The business process shown in the Figure 5.4 consist in the tasks needed to request for vacations inside a company, from the initial request up to its approval or rejection. The graphical notation used follows the concepts presented on the Section 2.2.1.

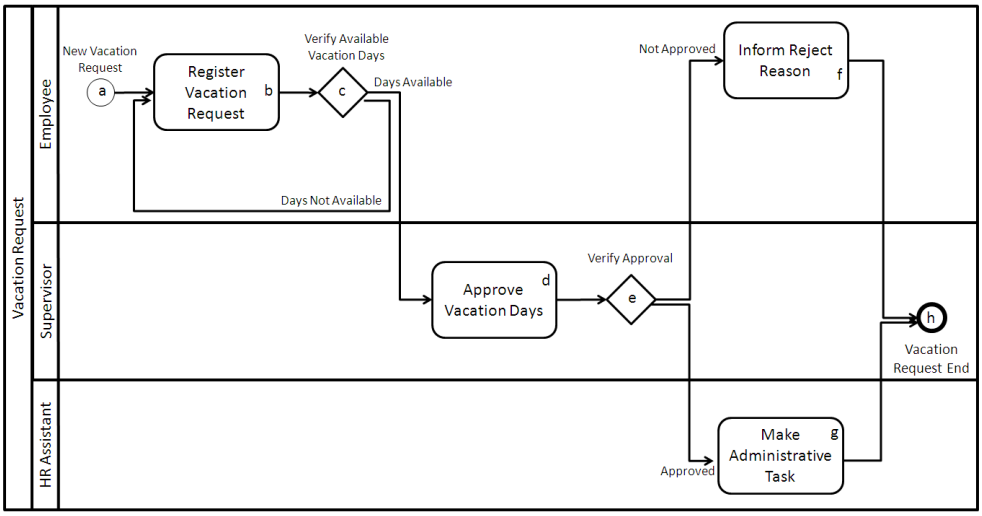

Figure 5.4: The *Vacation Request* business process.

The process of *Vacations Request* starts when any *employee* of the organization submits a vacation request, once the requirement is registered, the request is received by the immediate *supervisor* of the employee requesting the vacation, the supervisor must approve or reject the request, if the request is rejected the application is returned

to the applicant/employee who can review the rejection reasons. If the request is approved a notification is generated to the *Human Resources Representative*, who must complete the respective management procedures.

5.2.5 Applying the first step to the example

Starting from the *Vacation Request* example, in the first step, the *Input File Parser* should recognize the following things:

- **Vertexes**:
  - **a**: New Vacation Request
  - **b**: Register Vacation Request
  - **c**: Verify Available Vacation Days
  - **d**: Approve Vacation Days
  - **e**: Verify Approval
  - **f**: Inform Reject Reason
  - **g**: Make Administrative Task
  - **h**: Vacation Request End

  Note: The letters are just written now to facilitate the reading of the example (they are not meant to be included in the model). Besides, the order of the vertexes is not important.

- **Edges**:
  - From **a** to **b**
  - From **b** to **c**
  - From **c** to **b** (With condition 1 - See later)
  - From **c** to **d** (With condition 2 - See later)
  - From **d** to **e**
  - From **e** to **f** (With condition 3 - See later)
  - From **f** to **h**
  - From **e** to **g** (With condition 4 - See later)
  - From **g** to **h**

- **Parameters**: As mentioned before, the *parameters* are not part of the real *BPMN* but I will add this formality starting from now.

- *Description*: This is the parameter that has a textual description of the requested vacation done by the *Employee*. This parameter is being inserted in the task *B: Register Vacation Request* and is being read by the *Supervisor* in the task *D: Approve Vacation Days*.
- *Requested Working Days*: This is the parameter that contains the number of total requested days inserted by the *Employee*.
- *Available Vacation Days*: This is a value containing the rest of vacation days that are left for that year to the *Employee*.
- *Approval*: This is the result of the approval decision done by the *Supervisor*. This parameter can take two possible values: *Approved* or *Not Approved*.
- *Reject Reason*: This is the text inserted by the *Supervisor* in the case that he has rejected the vacation request. This value will be read later by the *Employee* in the task *F: Inform Reject Reason*.

- **Conditions**:
  1. If *Requested Working Days* are higher than *Available Vacation Days*
  2. If *Requested Working Days* are lower or equals than *Available Vacation Days*
  3. If *Approval* equals to *Not Approved*
  4. If *Approval* equals to *Approved*

5.3 Generating all possible paths

This section describes the second step of the *ReTB*.

This step consists in the reading of the *BPGraph* model and its exploration in order to obtain a collection of paths.

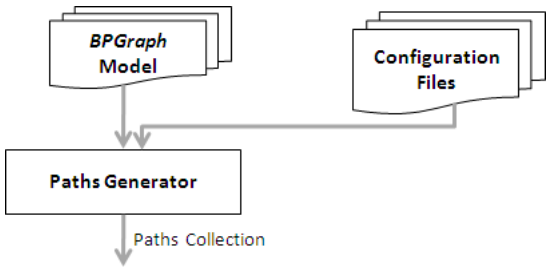

Figure 5.5: The second step of the *ReTB*: The *BPGraph* model is explored by the *Paths Generator*. The output is a collection of possible execution paths.

As exposed in Section 3.1, the *Graph Theory* has already analyzed this matter many times. As this was already studied, the decision taken was to use the existing algorithms to start with the explorations of paths inside a graph.

An important remark is the possibility of using different algorithms can be analyzed in order to obtain the more performance as possible.

5.3.1 The suggested exploration algorithm

The algorithm that will be suggested is the one done by R. Simões. In his paper called *APAC: An exact algorithm for retrieving cycles and paths in all kind of graphs* [Sim09]. The algorithm is shown in the Figure 5.6.

```
1.  P ← {s}
2.  PUSH(S,P)
3.  while STACK-EMPTY(S) ≠ TRUE
4.     do u ← TAIL(P ← POP(S))
5.        for each vertex {v} ∈ Adj(u)
6.           do if FIND-NODE(P,v) ≠ TRUE
7.                 then if {v} = {t}
8.                         then OUT-PATH(P,v,t)
9.                         else V ← ∅
10.                              V ← P ∪ {v}
11.                             PUSH(S,V)
12.                else OUT-CYCLE(P,v)
13.  P ← ∅
```

Figure 5.6: The APAC algorithm by R. Simões.

The references required to understand the APAC algorithm pseudo-code of the Figure 5.6 are:

- *PUSH:* This a Stack function; typically adds one object to the stack.
- *Adj:* Adjacency list.
- *STACK-EMPTY:* This is a Stack function; returns true if is empty.
- *TAIL:* This set function returns the last vertex.
- *POP:* Removes the element at the top of the stack.
- *FIND-NODE:* Returns true if an existing vertex already exists in a given set.

- *OUT-PATH:* Return a found path to some data structure or write it to disk.
- *OUT-CYCLE:* Return a found cycle to some data structure or write it to disk.
- *s:* Is the initial vertex.
- *t:* Is the final vertex.

Other proposals for the *Paths Generator*

The use of the *APAC* algorithm is not the only way to implement the *Paths Generator*. As exposed as *State of the Art* (Chapter 3), many other proposals have been done following different models. Each of them has some interesting ideas to

The approach given by Nayak *et al* [NS09] focuses on the path generation starting from activity diagrams. The main difference with the proposed *ReTB* model is that they focus on the generation of an intermediate model formed by minimal regions. These minimal regions are equivalent to the concept of control constructs exposed before (Section 5.3.2). This approached was not finally used by this proposal because many incongruences were found if mixing control constructs inside an activity diagram. For example, the minimal regions could be possibly nested (for example, a split-join together with a loop, where the loop goes outside the region of the split-join). This situation can lead us to an incorrect model or the skipping of some important paths in the diagram. Another problem with this paper was the lack of description of the algorithms to find the minimal regions inside the activity diagram. However, the proposed approach gives the required flexibility to generate test cases for each way of defining the coverage criteria.

On the side of *Graph Theory*, the found paper is restricted to find all the paths and cycles inside the graph. The proposal is to extend this *APAC* algorithm including the most control constructs as possible.

Finally, many other similar proposals have been exposed in the *State of the Art* (Chapter 3).

5.3.2 Identifying control constructs

In addition of having an algorithm to identify paths inside a business process, it is of great importance to offer an additional functionality: *the identification of control constructs inside the business process model.* By *control constructs* we can refer to the smaller structures into which we can subdivide the *Business Process Model.* For example *loops*, *parallel activities* or *split-join* regions.

The reason of offering this additional functionality is because once we have identified them, we can expand or restrict the generation of test cases depending the expected *coverage criteria* that wants to be used. For example, if we would like to

execute loops four times (because we specified it in the coverage criterion premises), the base path will be generated with the addition of four times repeating the tasks included in the loop region.The same would happen with the other control constructs, we could therefore have more control of the test case generation once we have identified them.

Many authors have defined different control constructs to be identified in the models as those cited in [XLL05], [CLLW05], [NS09].

The proposed control constructs exposed below are inspired in those mentioned by Nayak *et al* [NS09], but instead of using *Activity Diagram's* vocabulary, its adapted to the *Business Process Model* terms. They are also mixed with the terms defined in the *Workflow Patterns* (See Section 2.3 for details).

Cycles

A set of activities that can be executed repeatedly. The model can include either a *gateway*: making the flow divide in two flows where one of them returns to a previous activity; or not: a sequence flow going out from an activity directly to a previous activity. Even tough the first situation is considered as best practice, both situations can arise in business process models.

Exclusive choice

It is used to model two or more mutually exclusive alternative paths. Usually they are modeled by means of an *exclusive gateway*.

- **Merged**: The situation when, if x is an exclusive gateway, there exists a fork gateway such that all branches emanating from x merge into that gateway.
- **Un-merged**: The difference with the merged construct is that in this case there is no converging gateway at the end of the control flow. However, the control flow represented by this control construct is identified by knowing that each of the branches are terminated at some point.
- **K-out-of-N**: When k branches are merged and the remaining n-k are unmatched such that $k < n$.

Parallel split

Used to model two or more parallel execution paths. These parallel paths may require to be combined into a single flow. For this purpose a parallel gateway can be used either for starting the flows and also to close them.

- **Merged**: All the branches emerging from the parallel gateway joins together into another gateway.

- **Unmerged**: The different execution paths emerging from a parallel gateway may not converge into the same gateway. The identified construct can be identified if the different execution paths finishes at some point.
- **K-out-of-N**: The situation when a parallel gateway is not completely unmatched. Some of them may finish the flow independently from the ones that join together into another gateway.

Summary

After having defined the basic *control constructs* that can appear inside the business process model, it is being suggested the inclusion of a routine inside the algorithm exploring the paths of the business process. This is suggested because its important to give the flexibility of customizing the coverage criteria for each test case generation. In this way, each control construct can be managed independently from the others when generating the test cases.

The customization of how these control constructs can be found or managed is done by means of a configuration file. This is described in the following section.

5.3.3 Configuration File

To offer flexibility when exploring the business process model, is important to leave some options to be configured before executing the exploration. In this way, the algorithm should react differently depending the configured values and the test scenarios result should be different from one configuration to another.

Using the *APAC algorithm*, the detection of cycles is already included. Therefore, the *configuration file* is reduced to configure just the number of times a cycle is surpassed to generate test cases.

Notice that this case is the simplest possible, but this configuration file is meant to include many other parameters configuring how are the control constructs managed (e.g. *cycles*, *exclusive choices* and *parallel splits*).

5.3.4 Applying the second step to the example

The Figure 5.7 shows a simplified example of the different steps done by the APAC algorithm using the business process shown in the Section 5.2.4.

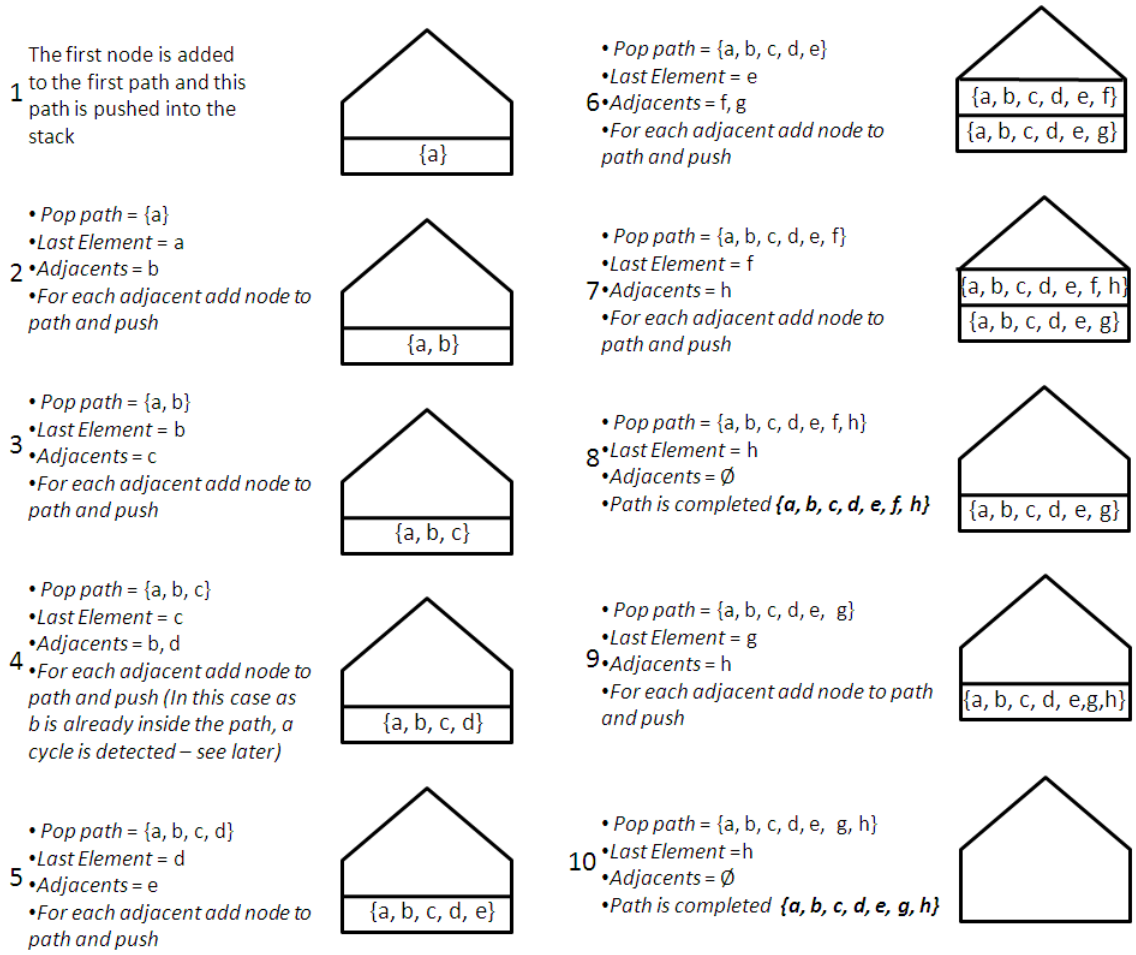

Figure 5.7: The steps of the APAC algorithm for the example in Section 5.2.4.

In any case, even if the APAC algorithm is not used and another is used in replacement, the following result should be given as output:

- *First Path*:
  - **a**: New Vacation Request
  - **b**: Register Vacation Request
  - **c**: Verify Available Vacation Days
  - **d**: Approve Vacation Days
  - **e**: Verify Approval
  - **f**: Inform Reject Reason

  - **h**: Vacation Request End

- *Second Path*:
  - **a**: New Vacation Request
  - **b**: Register Vacation Request
  - **c**: Verify Available Vacation Days
  - **d**: Approve Vacation Days
  - **e**: Verify Approval
  - **g**: Make Administrative Task
  - **h**: Vacation Request End

Notice that in this business process there are cycles inside. The previous expected result assumes that the *Properties File* was configured as default, passing just once through a cycle and not repeating the involved tasks. However, if this file was configured differently, the result should be slightly changed. For example, if the configured value says that the cycle should be passed until four times, the expected result would be one path for each case: the first passing once through the cycle, the second passing twice and so on. Another behavior could be to generate just one path for the configured value. That is, it would generate the same two paths as before, but repeating the tasks included in the cycle four times.

5.4 Retrieving test data

This section describes the third step of the proposed approach.

In this step, the data is being generated to be inserted later into the test cases.

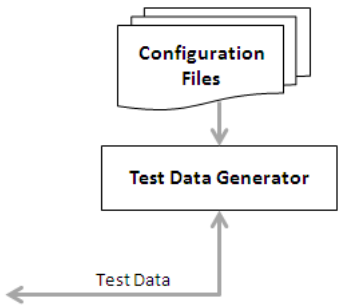

Figure 5.8: The third step of the *ReTB*: The test data is generated by the *Test Data Generator*.

The test data generation problem in this work is defined as follows: *for a given step involved in the process, find an input on which this step is executed.*

The responsibilities of the *Test Data Generator* are, given a parameter of the process inside a vertex (task, event or gateway), generate a testing value for it.

There are many different ways of doing this as it was mentioned in the *State of the Art* (Chapter 3). Therefore, the *Test Data Generator* can adopt any of those approaches.

The *ReTB* approach has the restriction of having the paths already generated as input. In other words, as the path is already given, the test data has to be adapted in order to match with the followed flow. Therefore the mentioned[1] *Path Oriented Test Data Generator* or either *Random Test Data Generation* are suitable for this approach.

In the model mentioned as *BPGraph* (Section 6.2.3) we consider the existence of *Parameters* and *Conditions*. This definitions can be present or not inside the incoming *Business Process Model*. As it was said, it would be highly recommended to add this kind of formalities inside a business process model to facilitate, among other things, the test case generation. The *Parameters* are defined by a name and a type whereas the *Conditions* are defined by one or more parameters, a logical operator and values. This information is needed for the generation of test data. In the case these formalities are not expressed, the *Test Data Generator* will not be executed and the *Test Cases Generator* will be reduced to the *Paths Generator* mechanism.

Notice that in the *BPGraph*, the *Parameters* are just defined by the type. But, it can be also possible to add conditions to it. This is, for example, a parameter called *Age* could have the type Integer and two conditions: 1-*Age minor than 100*; 2-*Age is higher than 0*. Another possibility is adding a matching condition that is the same as saying if *All* the conditions have to be met or *Any* of them.

Therefore, the proposed *Test Data Generator* will behave like this:

- **Tasks or Events**: When they are being evaluated, if they have a *Parameter*, the type will be checked and a value from that type will be retrieved. If there are conditions attached to the given *Parameter*, the retrieved value has to correspond to the condition so that evaluates true.

- **Gateways**: In the case of *Gateways*, the important matter is that the path is already given. So, it is needed to check which are the following tasks in the path. In this way we can know which value or *Condition/s* has/have activated that gateway in order to follow the corresponding path.

[1] *State of the Art* (Chapter 3)

5.4.1 The suggested data generation approach

As in the case of *Path Generation*, the *APAC* algorithm was suggested, in this case, the *Random Test Data Generation* will be mentioned as a possible approach. Remember that there is not one solution for the component *Test Data Generator* but it can be adapted to different necessities in different ways.

The *Random Test Data* approach generates test input values using a random number generator. Therefore, the proposal is to include some *Configuration Files* (that will be described in the next section) in order to have some testing values for each type of *Parameter* that we may have. For example, if our parameter is an *Integer*, the integer configuration file will be chosen and a random value from the file will be retrieved.

Pseudo algorithm for retrieving values

In the case of having *Conditions* restricting the values of the *Parameter*, after retrieving the random value, the conditions will be checked. If the conditions evaluate true, the value will be finally associated to the parameter. In the other side, if it evaluates false, another random value will be retrieved and checked. This procedure will continue until 1-The condition evaluates true; 2-A maximum of iterations has been reached. In the latter case, if we are treating with an Integer and the condition says *Age minor than 100*, the easiest way to generate this value is to substract a unit from the given top value. In this case, the retrieved value for the parameter *Age* will be *99*.

Therefore, the different types will have a slightly different behavior when they are being retrieved:

- *If there are no conditions*: A value from the corresponding type *Configuration File* will be retrieved.
- *If it has one or more conditions*:
  1. A value from the corresponding type *Configuration File* will be retrieved.
  2. The retrieved value will be checked among (*All* or *Any* as specified) the conditions.
     - If the *All* or *Any* of the conditions evaluate *true*, the value was found.
     - If they evaluate *false*, continue with the step 1 until:
       * The conditions evaluate true
       * A configured maximum of trial iterations is being reached (see next section for configuring this parameter)

5.4.2 Configuration File

The configuration file exist to make a personal configuration of the expected behavior of the *Test Data Generator*. Therefore, some attributes will be described to give some flexibility to the component.

For example, as said in the previous section, the *trial_iterations* parameter will configure the number of times the test data generator will produce random values when not finding the appropriate one.

The number of configuration parameters can be increased or modified to adapt the *Test Cases Generator* to each different need.

Within the *Test Data Generator*, other configuration files may be required. In the first proposal (exposed in the *Implementation* example), the configuration of possible values will be done manually: it will be one configuration file for each data type (e.g *text*, *numeric* and *date*). However, its recommended that this configuration is done automatically.

5.4.3 Applying the third step to the example

The result of the test data generation for the example given in the Section 5.2.4 should be as the one shown below. Notice that this results will be given for one of the paths and equivalently will happen to the other possible path.

- **a**: *New Vacation Request*: In this event there is no *Parameter* to generate test data.
- **b**: *Register Vacation Request*: The parameters being inserted in this tasks are:
  - *Description*: This is a free text type. This means that any kind of string of characters can be inserted. For example *I would like to take vacations as soon as possible*. There are no conditions associated to the parameter.
  - *Requested Working Days*: This is an *Integer* number. The retrieved value should be therefore an integer. If the parameter has a condition associated to it, the retrieved number has to agree with the condition.
- **c**: *Verify Available Vacation Days*: This gateway has a condition to decide which transition to follow. As in this path, the next task is *Approve Vacation Days*, the followed transition is the one that agrees with the result equals to *Days Available*.
- **d**: *Approve Vacation Days*: Two parameters are inserted in this task:
  - *Approval*: Two values could be inserted here, either *Approved* or *Not Approved*. As it will be seen in the next gateway, this parameter will finally take the value *Not Approved*.

  - *Reject Reason*: This is a free text type and any kind of string of characters can be inserted. For example *We are very hurried with the project, it is not possible now.* There are no conditions associated to the parameter.

- **e**: *Verify Approval*: This gateway is checking the value of the previous task: *Approve Vacation Days*. In this case, as the next task is *Inform Reject Reason*, the parameter involved *Approval* will take the value *Not Approved*. Notice that once one value is fixed for a parameter, all the other tasks have to see the same fixed value.

- **f**: *Inform Reject Reason*: This task carries the *Reject Reason* value to be shown in the task. It should be the same as the one inserted in the task *Approve Vacation Days*.

- **h**: *Vacation Request End*: In this event there is no *Parameter* to generate test data.

5.5 Constructing the test cases

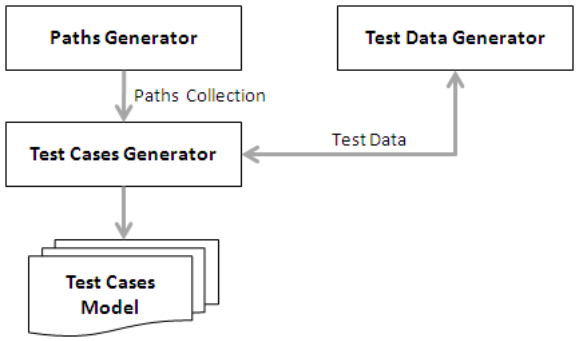

Figure 5.9: The fourth step of the *ReTB*: The test cases are generated combining the paths with the generated test data.

Finally, the *Test Cases Generator* will be the one that receives the collection of paths as input and explores them one by one in order to generate the *Test Cases*. While checking each *event*, *task* or *gateway*, it will call the *Test Data Generator* in different ways depending on the construct to get the test data values for each parameter (This was explained in detail in the Section 5.4).

At the end, the *Test Cases Generator*, will complete the different classes present in the *Test Cases Model*. This model can be, as well, done differently for each need that the *Test Cases Generator* may have.

The proposed *Test Cases Model* is an extended version of the one mentioned in the *Application Context* (Section 4).

5.5.1 The Test Case Model

The proposed model is the one mentioned by Fraternali *et al* [FT10], and shown previously in the Section 4.1. As all the other components presented inside the *ReTB* approach, this model can be interchanged by other or adapted to other needs.

This model was completed in the Master Thesis of Mondello [Mon10] resulting in the one shown below:

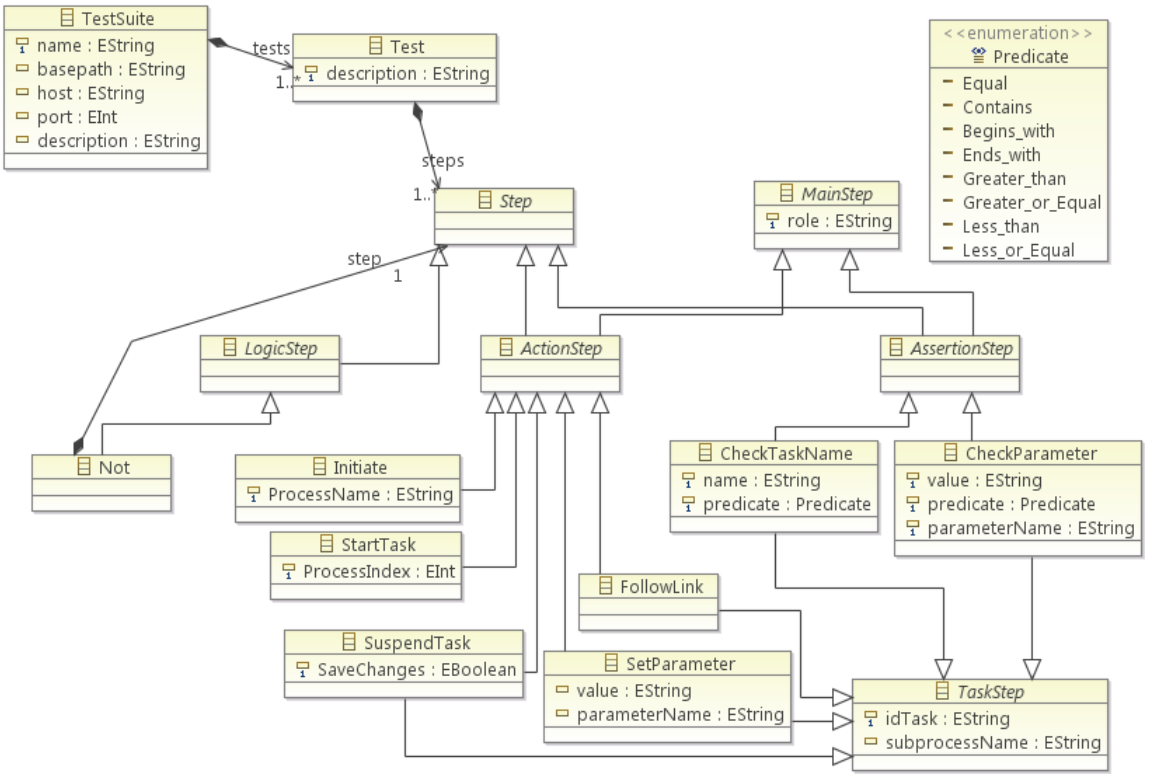

Figure 5.10: The complete *Test Case* model

This model was used in the context of *Multi-level testing for model driven web applications* and it belongs to a complete automatic testing engine described in the Section 4.1.

Basically, the model is done by a collection of steps that will be executed one after the other to emulate the execution of each task inside the business process.

Each proposed *Step* has a different purpose:

- *Initiate*: This step has the responsibility of activating a process. The parameters that has to be specified is the *ProcessName* that correspond to the *Name* attribute of the *Start Event* of the main model.
- *StartTask*: This step is the one starting a task inside an active process. Every user of the generated application will have a main page where he can see a list of the tasks that can be activated by him. To be able to distinguish among different processes, the attribute *ProcessIndex* was added to this step.
- *SetParameter*: This step assigns a value to a parameter. The parameter is identified by its *ParameterName*.
- *SuspendTask*: This step enables the user to suspend a running task. Depending on the value assigned to the *SaveChanges* attribute, the user can choose if saving the values that the parameters of the task had at that specific moment.
- *FollowLink*: This step finishes the running task and activates the next task of the flow being run.
- *CheckTaskName*: This step checks if the name of the executing task satisfies a certain predicate respecting to its *Name* attribute.
- *CheckParameter*: This step verifies that the parameter *parameterName* is the same to some other predicate value.

All the exposed steps are done within a task. As a consequence they carry a *idTask* and a *subprocessName* attributes (the latter attribute is optional).

For every step, a *role* attribute must be also specified. This attribute belongs to a *MainStep* that indicates the equivalent property *Role* that is defined for every *Lane* during the creation of a business process diagram.

5.5.2 Applying the fourth step to the example

As regards the example, mentioned firstly in the Section 5.2.4, the resulting collection of steps inside each test case should give this result:

- **Initiate** Role: Employee.
- **Start Task** Role: Employee.
- **Set Parameter** Task: *Register Vacation Request*; Parameter Name: *Description*; Value: *This is a random generated text*
- **Set Parameter** Task: *Register Vacation Request*; Parameter Name: *Requested Working Days*; Value: *30*

- **Follow Link** Role: Employee; Task: *Register Vacation Request*.
- **Start Task** Role: Supervisor.
- **Set Parameter** Task: *Approve Vacation Days*; Parameter Name: *Approval*; Value: *Not Approved*
- **Set Parameter** Task: *Approve Vacation Days*; Parameter Name: *Reject Reason*; Value: *This is a random generated text*
- **Follow Link** Role: Supervisor; Task: *Approve Vacation Days*.
- **Start Task** Role: Employee.
- **Set Parameter** Task: *Inform Reject Reason*; Parameter Name: *Reject Reason*; Value: *This is a random generated text*
- **Follow Link** Role: Employee.

5.6 Producing the output

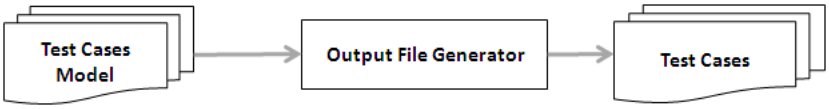

Figure 5.11: The last step of the *ReTB*: The test cases are stored in a file given as an output.

The last step of the *ReTB* proposes the generation of a file containing the *Test Cases*. The file is meant to be used later by another framework that will implement and execute the platform dependent test cases.

The *Output File Generator* can be implemented in many ways because each different *Test Case Generator* may need different kind of adaptation. In the following chapter, one implementation can be seen as an example.

5.7 Customizing the test coverage

As explained in the Section 2.1.2, the test coverage will give an idea of how complete the generated test suite is. For the purpose of this study, the defined coverage criteria will be as follows:

- *Path coverage:* How many paths of the business process are evaluated by the automatically generated Test Suite. By default, each path of the business process is mapped to one test case. In other words, all the paths will be covered by default.
- *Loop coverage:* How many times a loop is evaluated. By default, each loop structure will be followed once. There will not be additional test cases for each time the loop is reached (unless is configured differently).
- *Parallelism coverage:* How the parallel paths are evaluated. A parallel structure in *WebRatio* is symbolized as a *Parallel gateway* split and a *Parallel gateway* join. The default criteria is to evaluate randomly each task inside the parallel structure before arriving to the join gateway. Afterwards, the test will continue as usual.
- *Test data coverage:* How the test data is treated to cover the most cases as possible.

Other than using the default criteria, the option of varying the ways that the test suite covers the business process model will be supplied by a parameterizable configuration.

Chapter 6

Implementation

This chapter describes the implementation that was done satisfying the previously proposed design.

6.1 Overview

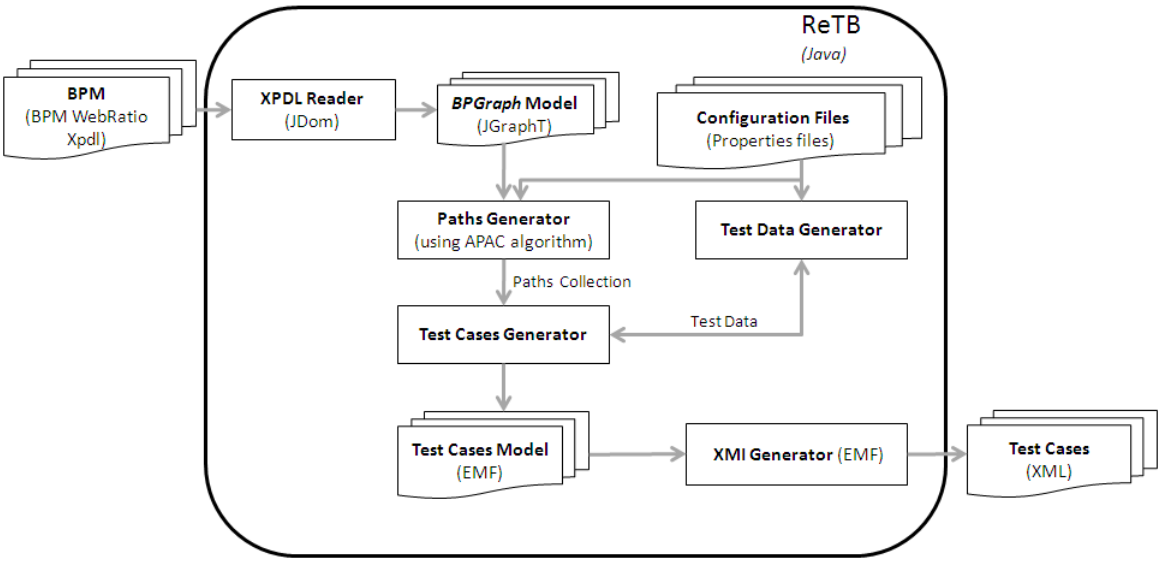

Figure 6.1: Overview of the proposed solution.

The implementation was done as a combination of existing *open source* languages and libraries, such as Java [GJSB05], JDom [Hun09], JGraphT [JGr10] and EMF [BBM03].

The overall idea is that after having modeled a business process in *Webratio*, make an export into an *XPDL* file[1], process this file to obtain the platform independent *Test Cases* and generate a final file comprising them.

The figure 6.1 summarizes the conceived solution. To start with, the business process diagram in form of XPDL (exported from *WebRatio*) is read by the *Xpdl-Reader* that is responsible of translate it into a graph structure. The library used to model the graph is *JGraphT*. Consequently, the *Paths Generator* is the one that, using the exploration algorithm, will generate all the possible paths that the business process may take. All these paths, together with some generated test data (that will be extracted by the *Test Data Generator*), will be complemented to generate the platform independent test cases. Finally, the *XMI*[2] *Generator* will produce a final file containing the generated test cases.

To finalize, its important to remember that any of the proposed components can be interchanged by other ones with different algorithms and/or technologies but respecting the same functionality. The shown solution is therefore a prototype and it can be modified and extended for obtaining tailored or improved results.

6.2 Capturing the model element details

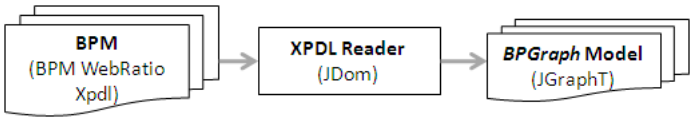

Figure 6.2: The implementation of the first step.

6.2.1 The XPDL file from *WebRatio*

The input consists on an XPDL file exported from *BPM WebRatio*[3]. This file contains all the information of the business process formatted with different tags than can be summarized as shown in the Figure below.

```
<?xml version="1.0" encoding="UTF-8"?>
<Package xmlns="http://www.wfmc.org/2008/XPDL2.1">
  <PackageHeader>
  ...
  </PackageHeader>
```

[1] See more about XPDL in Section 2.2.3

[2] *XML Metadata Interchange.* It is a standard which specifies the storing method of a UML model in an XML file.

[3] See more about WebRatio in Section 4.2

```
  <Pools>
    <Pool Id="" Name="" BoundaryVisible="" Process="">
      <Lanes>
        <Lane Id="" Name="" ParentPool="">
          ...
        </Lane>
        ... Other lanes
      </Lanes>
    </Pool>
  </Pools>

  <WorkflowProcesses>
    <WorkflowProcess Id="" Name="">
      <ActivitySets/>
      <Activities>
        <Activity Id="" Name="">
        <Documentation>...</Documentation>
        <Event>
            <StartEvent Trigger=""/>...  or...  <EndEvent/>
        </Event>
        <Route GatewayType=""/>
          ...
        </Activity>
        ... Other activities
      </Activities>
      <Transitions>
        <Transition Id="" From="" To=""/>
        ... Other transitions
      </Transitions>
    </WorkflowProcess>
  </WorkflowProcesses>
</Package>
```

The needed information for the aim of this work is detailed below divided by which will be the component using this information.

Information needed for the paths generation

- **Lanes**: It will be considered to be just one pool[4] inside the business process model. This pool can contain several *Lanes*. The needed information for describing the lanes is:
  - *Id:* The id of the lane.
  - *Name:* The name of the lane corresponds to the name given to the *Role*.
- **Tasks**: From each task, the important information to be extracted is:
  - *ActivityId:* The id of the task.
  - *Name:* Name of the task.
  - *IdLane:* The id of the lane where the activity is placed. This will give the particular *Role* that will be able to perform the task.

[4]The existence of just one pool inside a business process model can be reconsidered in the future.

- **Transitions**: From each transition, the important information to be extracted is:
  - *TransitionId:* The id of the transition.
  - *From:* The id of the source activity.
  - *To:* The id of the target activity.

Information needed for the test data generation

- **Parameters**: The parameters are defined globally for the process. Certain activities will have some parameters attached that will correspond to those that the activity can modify (Marked as *editable*). Each parameter will have:
  - *Name:* Name of the parameter.
  - *Type:* Data type of the parameter (The accepted types in *Webratio* are *boolean, date, decimal, file, integer, password, string, text, time, timestamp, url*).
- **Tasks**: From each task, the important information to be extracted are the parameters that are treated within the task (These are just those marked as *Visible* and *Editable*. The attribute **Name** will identify this parameter as defined globally.
- **Gateways**: As detailed in Section 4.2.1, the *Gateways* have a different way of being configured within *WebRatio*. Summarizing, the gateway can be of different *ExecutionType*. For each type, the more relevant information is described:
  - **User:** The attribute that can help the test automation is the user *Role*. This attribute corresponds to the *Name* of the *Lane* which *IdLane* is associated to the gateway.
  - **Service:** It is important to know how the services will be called and which are the respective *input* parameters.
    - *Expression:* The information regarding each possible target and its corresponding logic expression is needed. For example, if the expression is $Age \geq 20$ to follow the *TargetActivityId=1* (and the rest to follow a *TargetActivityId=2*), this attributes will be needed:
      TargetActivityId: 1 (This will be the target activity if the following expression evaluates true).
      - *ParameterName*: Age
      - *LogicOperator*: $\geq$

- *Value*: 20

  TargetActivityId: 2 (This will be the other possible target).
- *ParameterName*: Age
- *LogicOperator*: <
- *Value*: 20

* *Literal*: For each of the subsequent flow branches, the corresponding *TargetActivityId* attached to one of the literal *values*. Following the example seen on the Figure 7.12:

  TargetActivityId: 1
- *ParameterName*: Approval
- *Value*: *Approved*

  TargetActivityId: 2
- *ParameterName*: Approval
- *Value*: *Not Approved*

Another thing to note is that the conditions within a target can be many. As will be shown in the Figure 7.10, the conditions can be also combined by specifying a matching parameter. The matching can take two values: *All* (All the conditions must evaluate true) or *Any* (One of the conditions must evaluate true).

Changes request for the *WebRatio* exporter

The used version of *WebRatio* is the *WebRatio Enterprise 6.0.0Beta*. This version, as seen before, has an XPDL export with some basic information.

For the purpose of this work, a modification of the generated XPDL is required. The modification should include all the attributes that were described in Section 6.2.1 and in Section 6.2.1.

The resulting XPDL format should look as the one shown below if the same implementation of the XPDL Reader wants to be used in the future.

```
<?xml version="1.0" encoding="UTF-8"?>
<Package xmlns="http://www.wfmc.org/2008/XPDL2.1">
  <PackageHeader>
  ...
  </PackageHeader>
  <Pools>
    <Pool Id="" Name="" BoundaryVisible="" Process="">
      <Lanes>
        <Lane Id="" Name="" ParentPool="">
          ...
        </Lane>
        ... Other lanes
      </Lanes>
    </Pool>
```

```
  </Pools>
  <Parameters>
    <Parameter Name="" Type="" />
    ...Other parameters
  </Parameters>
  <WorkflowProcesses>
    <WorkflowProcess Id="" Name="">
      <ActivitySets/>
      <Activities>
        <Activity Id="" Name="" LaneId="" ExecutionType="">
            <Documentation>...</Documentation>
            <Event>
                <StartEvent Trigger="" />...  or...  <EndEvent/>
            </Event>
            <Route GatewayType="" ConditionType="">
                <Target TargetActivityId="" Matching="" />
                    <Condition ParameterName="" LogicOperator="" Value="" />
                    ... Other conditions
                </Target>
                ... Other targets
            </Route>
            <Parameters>
                <Parameter Name="" />
                ... Other parameters
            </Parameters>
          ...
        </Activity>
        ... Other activities
      </Activities>
      <Transitions>
        <Transition Id="" From="" To="" />
        ... Other transitions
      </Transitions>
    </WorkflowProcess>
  </WorkflowProcesses>
</Package>
```

Notes about the expected XPDL file for each basic construct:

Parameters

- The *Type* of the parameters accept as values either *string* or *number*. Any other type will be recognized later as *string*.
- If one of the parameters in *WebRatio* refers to a *Business Object*, this will result in one parameter *xpdl line* per each parameter inside the business object.

Activities

The activities structure contains information for *Tasks*, *Gateways* and/or *Events*. Each of them will be treated differently.

- *Tasks*

  - The *Parameters* sub-structure within the *Activity* structure is only used by *Tasks*.
    * The *Name* of the parameter has to be the same as the global parameter configured in the *Parameters* structure, named before.
    * The parameters to be added in the activities are just the ones marked in *WebRatio* as *editable*.
  - The values of the *ExecutionType* attribute can be: *Manual*, *Service* or *User*.

- *Events*
  - The *Event* sub-structure within the *Activity* structure is only used by *Events*.
    * Inside, it can contain either another sub-structure called *StartEvent* or *EndEvent*, to differentiate both types of events.

- *Gateways*
  - The *ExecutionType* attribute can get the values: *Service* or *User*.
  - The *Route* sub-structure iwithin the *Activity* structure is only used by *Gateways*.
  - The *ConditionType* attrubute of a *Route* can get the values: *Expression* or *Literal*. Where:
    * *Expression* must have the attributes *TargetActivityId*, *ParameterName*, *LogicOperator* and *Value* inside the corresponding *Condition*s.
    * *Literal* its just necessary to add the attributes *TargetActivityId*, *ParameterName* and *Value*. The number of *Conditions* inside a *Literal* is just one.
  - The *Matching* parameter inside a condition can take two values *All* or *Any*. This parameter is necessary just if the *ConditionType* value is *Expression*.
  - The values that a *LogicOperator* may take are restricted to: *lower*, *lowerOrEquals*, *higher*, *higherOrEquals*, *equals*, *different* (This is because, the corresponding symbols ($<$, $\leq$, $>$, $\geq$, $=$, $\neq$) may affect the structure and reading of the XPDL file).
  - Within a *Route*, the *GatewayType* is today empty by default if the gateway is an *Exclusive Gateway*. It may be more clear to write the type even in this default case.

6.2.2 The XPDL Reader

The library used to read the exported XPDL file is *JDom* [Hun09].

The *XpdlReader* does an operation for each type of structure found in the XPDL (named as, for example *processActivity* and *processTransition*). Each of these operations will be the ones responsible of transforming the found information into a well known data structure. This data structure is explained in detail in the Section 6.2.3).

In the next example it can be seen how, using *JDom*, the *Parameters* structure can be read from the XPDL file and converted into the proposed *BPGraph*.

```
/**
 * Parses the parameters
 *
 * @param parameters
 */
private void parseParameters(List parameters) {
    for (int i = 0; i < parameters.size(); i++) {
        Element parameterElement = (Element) parameters.get(i);
        String name = parameterElement.getAttributeValue(XpdlTags.NAME);
        String type = parameterElement.getAttributeValue(XpdlTags.TYPE);
        MyParameter parameter = new MyParameter(name, type);
        graph.getParametersHashMap().put(name, parameter);
    }
}
```

6.2.3 The *BPGraph* model

After describing the proposed model shown in the Figure 5.3, the implementation was done adding the library provided by *JGraphT* [JGr10]. This library, apart from providing a basic graph data structure (formed by *vertexes* and *edges* connecting them), offers a basic set of algorithms to obtain information from any kind of graph (to get a specific node, given a node to get all its subsequent nodes, to get the shortest path between two nodes, etc).

The decision of using this library was mainly to take advantage of an already stable data structure and because the interface is simple, useful and extensible.

To acquire the best flexibility and control power, the basic classes of the *JGraphT* data structure were extended, such as those defining the *Graph*, *Vertex* and *Edge*.

The Figure 6.3 shows the final implemented *BPGraph* model.

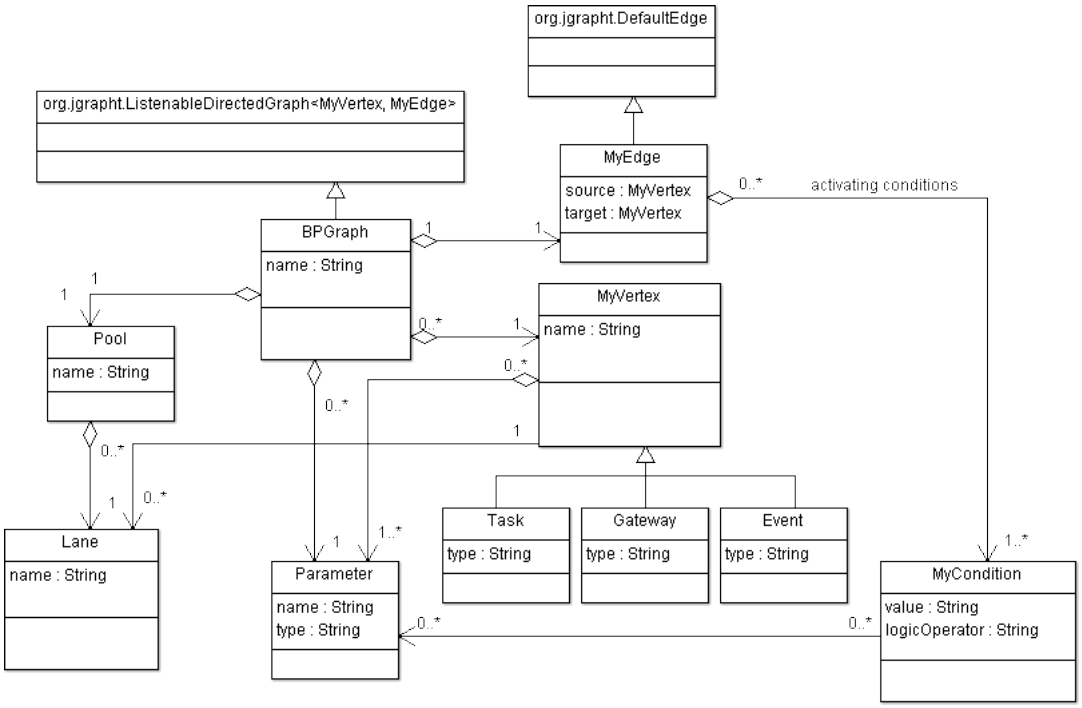

Figure 6.3: The *BPGraph* structure extending the one provided by JGraphT.

Concluding, to parse the incoming XPDL coming from *WebRatio*, a *XPDLReader* class was created which main functionality is to read the XPDL and transform it to a graph. The used libraries where *JDOM*[Hun09] and *JGraphT* [JGr10].

6.3 Generating all possible paths

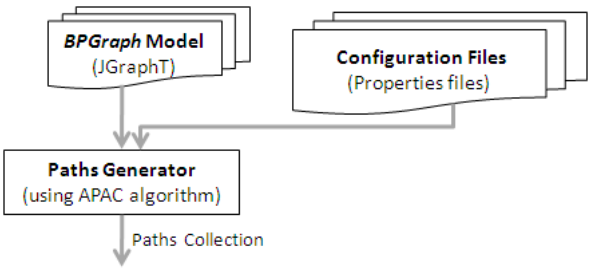

Figure 6.4: The implementation of the second step.

The resulting algorithm using *Java* and the *JGraphT* library is shown below. This algorithm shows the basic APAC algorithm that will be later completed to be able to customize the test case generation.

```
/**
     * This is the APAC algorithm to obtain all the paths (and cycles) between
     * two vertexes of a graph
     *
     * @param graph
     * @param start
     * @param end
     */
    public void findPathsFromTo(MyVertex start) {
        Stack<ArrayList<MyVertex>> stack = new Stack<ArrayList<MyVertex>>(); //↩
            Stack
        MyVertex u;
        ArrayList<MyVertex> p = new ArrayList<MyVertex>();
        int maxCycles = propertiesReader.getNumberOfClycles();
        HashMap<String, Integer> vCycles = new HashMap<String, Integer>();

        p.add(start);// 1
        stack.push(p);// 2
        while (stack.size() > 0) {// 3
            p = stack.pop();// 4
            u = p.get(p.size() - 1);// 4
            List<MyVertex> adj = Graphs.successorListOf(graph, u);// 5
            for (MyVertex v : adj) {// 5 For each successor
                if (!isCycle(p, v)
                        || cycleLessThanMax(p, maxCycles, vCycles, v)) {//If didn't↩
                             reach configured maximum number of cycles
                    if (isCycle(p, v)) {//If there is a cycle
                        addToCyclesCounter(vCycles, v); //Increases the cycle ↩
                            counter for that vertex
                    }
                    if (endVertexes.contains(v)) {// 7 Path reached last vertex
                        p.add(v);// 8
                        allPaths.add(p);// 8
                    } else {// Normal Vertex
                        ArrayList<MyVertex> newP = new ArrayList<MyVertex>();// 9
                        newP.addAll(p);// 10
                        newP.add(v);// 10
                        stack.push(newP);// 11
                    }
                }
            }// End of successors analysis
            p = new ArrayList<MyVertex>();// 13
        }
    }
```

The reference number and letters correspond to the APAC algorithm proposed steps as seen in Section 5.3.1.

As it can be observed, the algorithm was slightly modified in order to give the possibility of configuring the number of times the cycles are surpassed. This algorithm could be modified again in order to find different control constructs as suggested in Section 5.3.2, or to fit the functionality to the one required by the particular *Test Case Generator*.

6.3.1 Configuring the *Paths Generator*

Some decisions can be done when exploring the paths that will involve different delicate structures that have to be seen carefully:

Cycles

The existence of cycles within a business process has to be treated differently from the normal flow cases. This is because, depending on the chosen *Coverage Criteria*, it may be needed to execute it none, once, twice or n times.

Therefore, in the implemented solution, the number of cycles per *Test Case* will be configured in the *Properties file* (see below) using the attribute *number_of_cycles*. This file can be changed before executing the *Test Case* generation and will have the following behavior:

If the configured number of maximum cycles is *five*, there will be one test case for each possibility until reaching *five* cycles. In other words, the first test case will not pass through the cycle, the second will pass just once, and this repeats until the last generated test case passes *five* times. (So, the generated *Test Suite* will contain all those *Test Cases*).

This is the properties file used to configure the paths generator:

Listing 6.1: The *Paths Generator* properties file

```
number_of_cycles=0
```

In this case, the configuration file is set as default, it will not re-enter to the cycle. The tasks among the cycle chain will be surpassed just once.

6.4 Retrieving test data

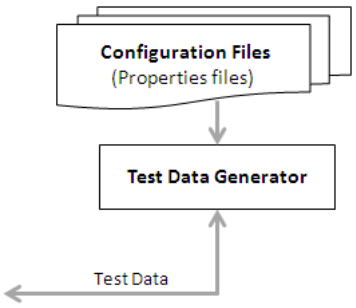

Figure 6.5: The implementation of the third step.

The chosen implementation is done in a simple way: The *Test Data Generator* will be asked for a random value for a certain type. Depending that type, the *Test Data Generator* will search in the appropriate *Configuration File* and return that value. In the case of having conditions associated to the expected value, it will generate random values (extracted from the same file) until the conditions are satisfied. If the conditions are not satisfied, it will continue trying a specified number of times and if that value is still not being found, generate it by itself.

As explained in Section 5.4, the data is generated after having the paths of the business process ready. As a consequence, the generated values will be retrieved depending of the followed path. The implemented solution was thus done differently for each gateway construct present in the *WebRatio* business process model:

Literal Gateway

The *literal gateways* are used when the flow will depend on a literal value that a parameter can take.

Repeating the example seen until now, the flow may take a different way depending of the current value of the *Approval* parameter.

- If the value is *Approved*, the flow will continue to the *Make Administrative Task*.
- In the other case, the value is *Not Approved* so the flow will go to the *Inform Reject Reason* task.

The *Test Data Generator* is executed after having a complete path inside the business process. As a consequence, when finding a *literal gateway*, the system checks the next node in the path to see which is the value that the parameter should have taken to arrive there.

For the example given, if the next node correspond to the *Inform Reject Reason* task, it is that the value that the *Approval* parameter has to have taken is *Not Approved*.

The gateways that may use *literals* for taking flow decisions are all of them except the *Parallel Gateway* (the parallel gateway does not need to take any decision to continue to the following nodes)

Expression Gateways

In the case of *Expression Gateways*, the flow continues different paths depending if one or more conditions are satisfied.

As seen in the example, the decision of the flow is this time associated to one condition.

- If the value of the parameter *Number of Vacation Office Days* is inferior to 30, then the flow continues to *Approve Vacation Days*.

On the other hand, if the value of the parameter *Number of Vacation Office Days* is superior or equals to 30, then the flow continues to Register Vacation Days.

- *All*: All the conditions must be satisfied.
- *Any*: One or more conditions can be satisfied.

The *Test Data Generator* is executed after having a complete path inside the business process. As a consequence, when finding an *expression gateway*, the system checks the next node in the path to see which is the value that the parameter should have taken to arrive there.

Following the example, if the next node in the path is *Approve Vacation Days*, this means that the condition *Number of Vacation Office Days is inferior to 30* has been satisfied. Therefore, the generated value has to be inferior to 30.

The test data generation algorithm will:

- If the *Properties File* was configured correctly, search among the inserted possible values randomly until getting one value that satisfies the condition. If not, continue generating random values from the file until a specific threshold (configured in the properties file as *trial_iterations*.
- If there is no configured value that can satisfy the condition, the value will be generated automatically (and, of course, satisfy that condition).

The conditions within an expression can be one or more than one, and they have to be satisfied depending on the selection *All* or *Any* (As will be shown in the Figure 7.10). Therefore, if the value is:

- *All*: The test data generator will do the same as explained before until satisfying all the conditions.
- *Any*: At this moment, the *Any* option is behaving as the same as the *All* option. This is because *Any* is included in the *All* condition. This can be changed in the future if needing a more detailed test data generation.

The gateways that may use *expressions* for taking flow decisions are all of them except the *Parallel Gateway* (the parallel gateway does not need to take any decision to continue to the following nodes)

6.4.1 Configuration File

There are two kind of configuration files to the *Test Data Generator*.

The main configuration file will consist in the collection of parameters affecting the logic of the test data generator.

At this moment the possible configurations include:

- *trial_iterations*:The value contained here represents the number of iterations that the system will do generating random values (taken from the properties file) until one logical condition evaluates positive. If the values are not well configured or forgotten in the file, the system will stop randomizing the values inside the file and try to generate one of its own (that complies with the given condition).

This is an example of a properties file used to configure the test data generator:

Listing 6.2: The *Test Data Generator* properties file

```
trial_iterations=10
```

On the other side, there are one file for each *Parameter* type inside the process. As a consequence, one file will be created with *numeric* values, other with *dates*, other with *strings*, etc.

These *configuration files* can be done as well in many different ways. For example they could be done as part of the BPM model, this is, for each parameter of the model, include some testing values.

Below, the used configuration files within this implementation example are being shown.

Listing 6.3: Example of *String* values properties file

```
string=String example, Random String, String result, Value of a String
```

Listing 6.4: Example of *Dates* values properties file

```
date=01/02/03, 02/02/02, 01/04/02
```

Listing 6.5: Example of *Number* values properties file

```
number=1,2,3,4,5,6,7,8,9,10,11,12,49,50
```

Listing 6.6: Example of *Text* values properties file

```
text=this is a long description, this is another description, this is an example of↩
     long texts, text example
```

These shown configuration files are the simplest and fastest way to generate random test data values within the prototype. However, there are many automatic generators of data values around. To find the appropriate one is recommended.

6.5 Constructing the test cases

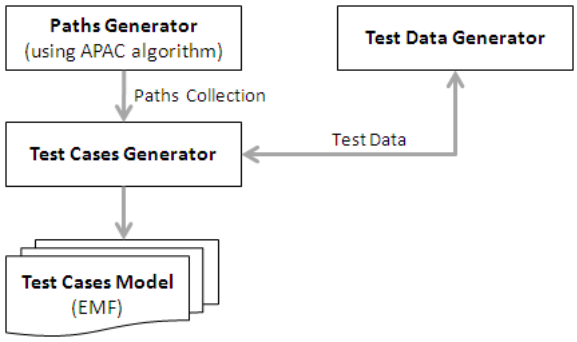

Figure 6.6: The implementation of the fourth step.

The final implementation of the *Test Cases Generator* is done by connecting all the corresponding components together. This is done by a Facade class called *MyTestCaseGeneratorFacade*. This class responsibility is to call the components passing their corresponding input and outputs. This can be seen in detail in the Figure 6.7.

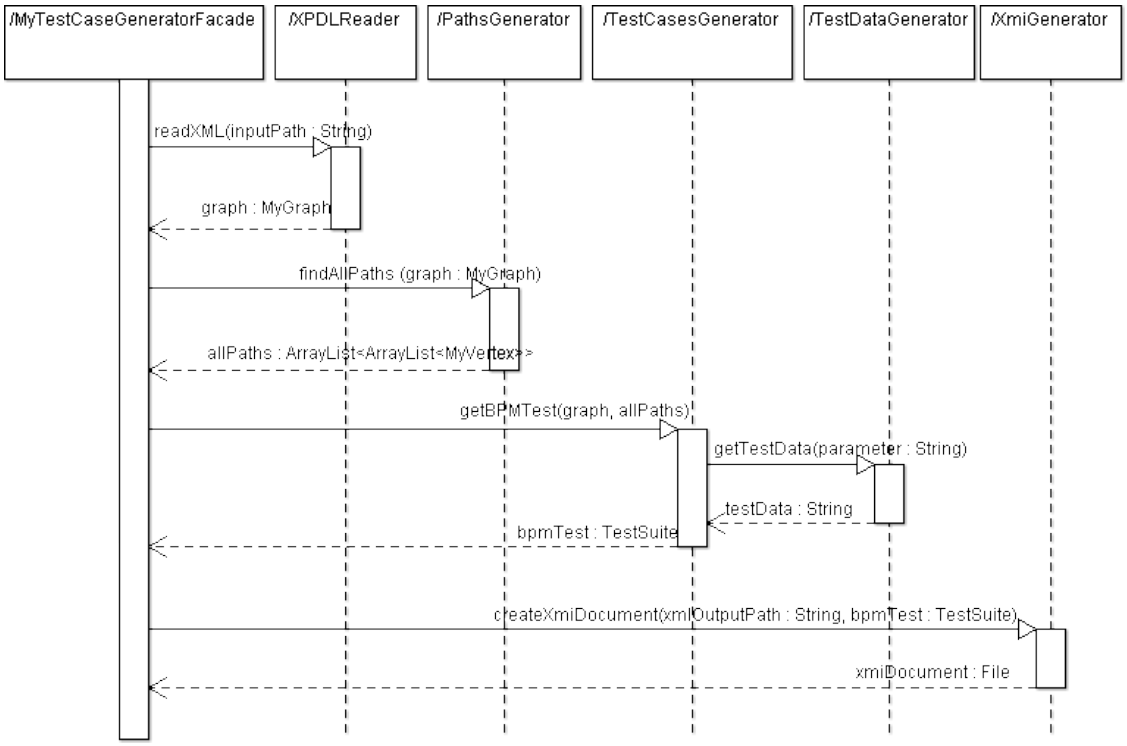

Figure 6.7: Sequence diagram containing the implemented classes interaction.

The class *TestCasesGenerator* is the one that will receive as input all the generated paths collection and give as an output the *Test Suite* (collection of *Test Cases*).

6.6 Producing the output

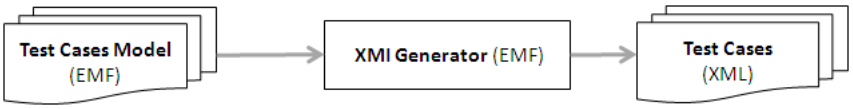

Figure 6.8: The implementation of the fifth step.

Once we have all the information we need to complete the platform independent Test Cases, we need to express them according to a specific model. To be able to do so, the *Eclipse Modeling Framework EMF* [BBM03] was chosen to facilitate the process

of generating an XMI[5] specification according to a *Test Cases* predefined model. In this case, the chosen model was already shown in the Figure 5.10.

6.6.1 Use of EMF

The *Test Cases Generator* proposed implementation uses EMF [6] in this way:

1. Creation of the *Ecore* model (as shown in the Figure 5.10).
2. Generation of *Java* code from the *Ecore* model.
3. Filling the Test Cases instances into those generated *Java* classes.
4. *XMI* serialization.

6.7 Summary of implemented components

The implemented components of the *Test Data Generator* and the technologies used are:

- **Input File Parser:** Converts the *XPDL* into the *BPGraph* structure.
  - It was implemented using *Java* and *JDOM*.
- **BPGraph model:** Classes representing either a *Business Process* as a *Graph*. The model was shown in the Figure 6.3
  - Implemented using *Java* and the *JGraphT* libraries.
- **Paths Generator:** Contains the algorithm to retrieve all the possible paths of a graph (Using the proposal of R. Simões [Sim09]).
  - Implemented in *Java*.
  - Its configuration can be done by means of a *properties file*.
- **Test Data Generator:**. Contains the basic functionality for retrieving test data for each parameter of the business process.
  - Implemented in *Java*.
  - Its configuration can be done by means of a *properties file*.

[5] *XML Metadata Interchange*. It is a standard which specifies the storing method of a UML model in an XML file.

[6] See more about this framework in the Section 2.5

  - The test data values are inserted manually in different *properties files*. Each file corresponds to a certain data type (*string*, *text*, *data* or *numeric*).
  - The test data values are retrieved in a random way.

- **Test Cases Generator:** Explores the collection of paths, and for each parameter calls the *Test Data Generator* to retrieve the corresponding values for the parameters of each task. Finally, it populates the *Test Case Model* classes.
  - Implemented in *Java*.
- **Test Cases Model:** The model that includes all the test cases steps, model proposed by Mondello [Mon10].
  - Implemented using *EMF*, starting from the proposed *Ecore* (shown in Figure 5.10).
- **Output File Generator:** Generates the resulting *XML* file containing the test cases.
  - Implemented using *Java* and the *EMF* libraries.

Chapter 7

Experimental results

7.1 The BPM taken as an example

To test the results of the implementation, a process named *Vacation Request* was used.

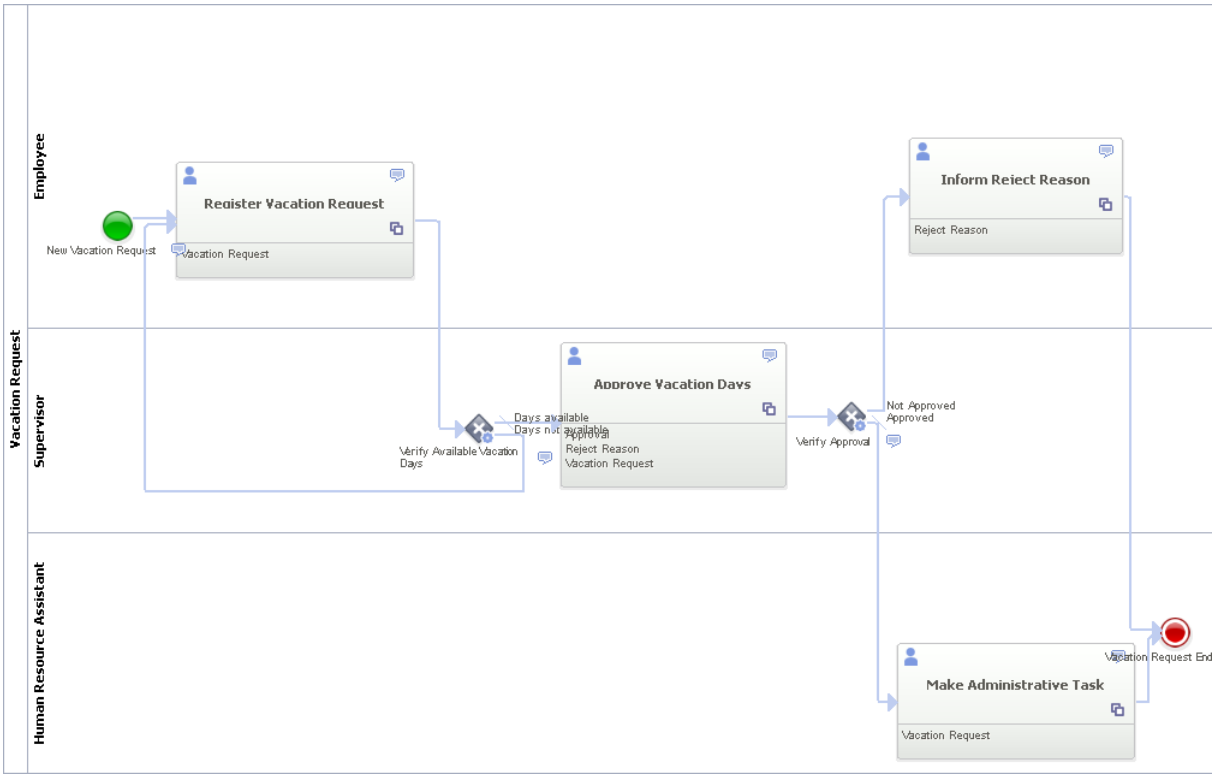

Figure 7.1: The *Vacation Request* example modeled in *WebRatio*

This process focuses in controlling the requests of Vacations of the organization, from the initial request up to its approval or rejection.

The process of *Vacations Request* starts when any *employee* of the organization submits a vacation request, once the requirement is registered, the request is received by the immediate *supervisor* of the employee requesting the vacation, the supervisor must approve or reject the request, if the request is rejected the application is returned to the applicant/employee who can review the rejection reasons. If the request is approved a notification is generated to the *Human Resources Representative*, who must complete the respective management procedures.

7.1.1 Roles

There are three roles defined for the process:

- *Employee:* This role represents those who will be able to request for the holidays.
- *Supervisor:* This role represents those who will be able to make the approval or rejection of the requested holidays.
- *Human Resource Assistant:* This role represents those who will be able to make the administrative tasks related with the requested holiday.

7.1.2 Business Objects

The created *Business Object* to be treated during the business process is the one called *Vacation Request*.

The attributes associated with the *Business Object* are:

- *Request date:* The date when the employee does the request for his holidays.
- *Employee name:* The name of the employee.
- *Start date:* The date when the requested holidays should start.
- *End date:* The date when the requested holidays should end.
- *Number of vacation office days:* The total amount of office days that the employee would be on holidays.

This business object can be configured in *WebRatio* as shown in the Figure 7.2.

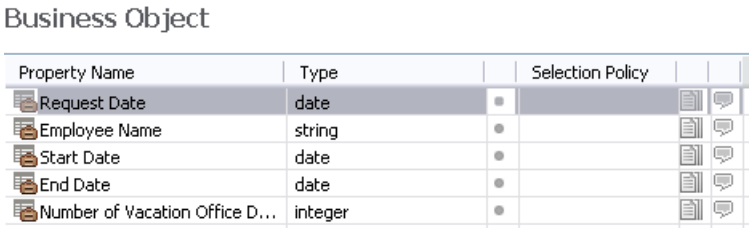

Figure 7.2: The modeled Business Object *Vacation Request*

Within the customizable attributes that are of interest for the purpose of this work, the *Case Values* corresponds to a restricted set of values for the property (shown in the Figure 7.2 as a grey point) and the *Selection Policy* is the execution-time cardinality (either single or multiple) of the values carried by parameters associated to the business object property.

7.1.3 Process Parameters

The parameters used in this business process are:

- *Vacation Request:* The business object as specified before (Section 7.1.2).
- *Approval:* The two values that can take the approval *Approved* or *Not Approved*.
- *Reject Reason:* The reason of the rejection written by the supervisor.

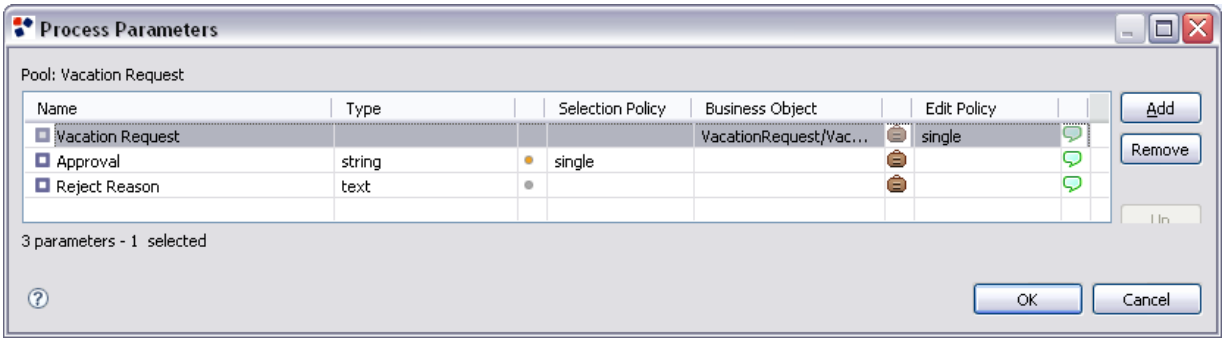

Figure 7.3: The process parameters of the *Vacation Request* business process

The dialog shows the current list of parameters for the the process. For each parameter it is possible to change the *Name* and *Type*. A restricted set of *Case Values* can be defined clicking on the circle icon which opens an external dialog.

If one or more case values is defined, the next property, named *Selection Policy*, is then enabled, thus allowing to define the execution-time cardinality (either single or multiple) of values carried by the parameter. As an alternative to plain types, a parameter can be associated to a *Business Object* using a specific selection dialog; the next property, named *Edit Policy*, is then enabled allowing to define how many instances can be edited at the same time. Clicking on the rightmost icon allows to define an *Analyst Note* for the parameter using a specific dialog.

7.1.4 Tasks

Register Vacation Request

This Task allows the *Employee* to define all the details of the *Vacation Request*.

The required parameters of this activity are shown in the Figure 7.4. In this figure we can notice that the employee has to fill all the values for the parameters corresponding to the *Vacation Request* business object.

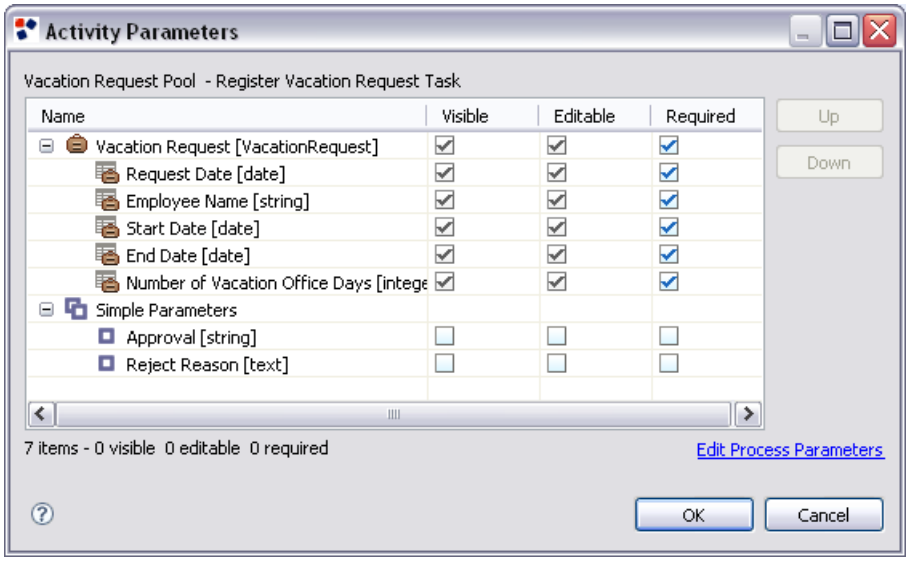

Figure 7.4: The parameters of the *Register Vacation Request* activity

Approve Vacation Days

This Task allows the *Supervisor* to decide to approve or not the *Vacation Request*.

The parameters that are used within this activity are shown in the Figure 7.5. In this figure we can notice that the supervisor can see all the information filled by the employee and is only enabled to modify the approval parameter that accepts only the

Approved/Not Approved values (as seen in the Figure 7.3). Besides, he can also fill a comment for explaining the decision taken.

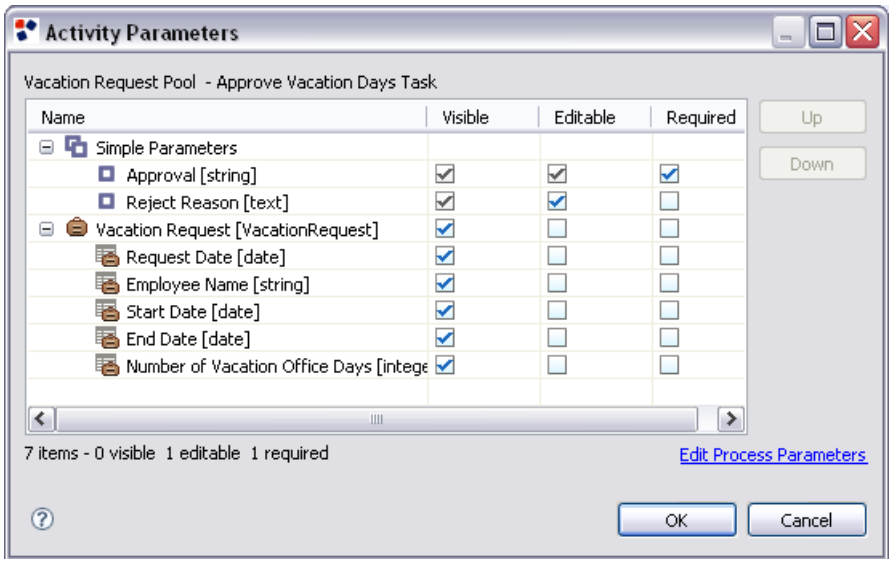

Figure 7.5: The parameters of the *Approve Vacation Days* activity

Make Administrative Task

This Task allows the *Human Resource Assistant* to perform the administrative task in order to register the *Vacation Request* if it has been approved by the *Supervisor*.

The required parameters of this activity are shown in the Figure 7.6. In this figure we can notice that the human resource assistant can see all the information regarding the vacation request done by the employee. There is no other detail regarding this activity. Probably, all the logic needed in this activity is not relevant for the purpose of the example.

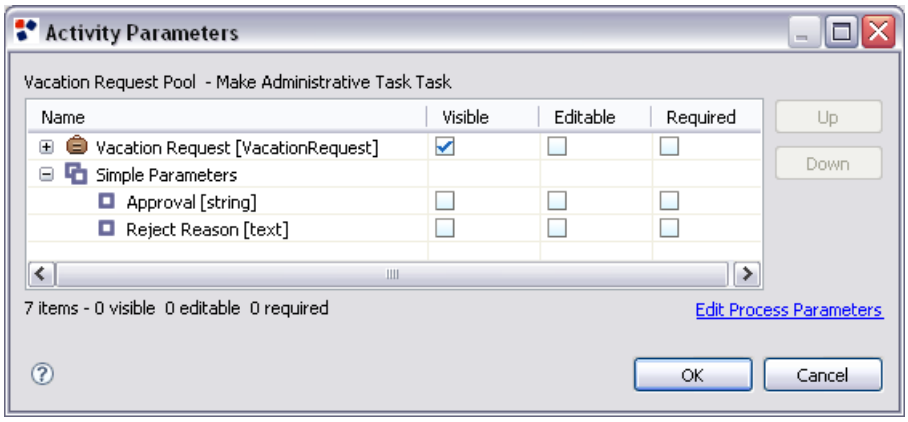

Figure 7.6: The parameters of the *Make Administrative Task* activity

Inform Reject Reason

If the *Vacation Request* has been rejected, this Task informs the *Employee* of the *Reject Reason*.

The required parameters of this activity are shown in the Figure 7.7. In this figure we can notice that the employee can only see the reject reason. Obviously, if he is able to access to this task, it already means that the value of the approval parameter is *Not approved*.

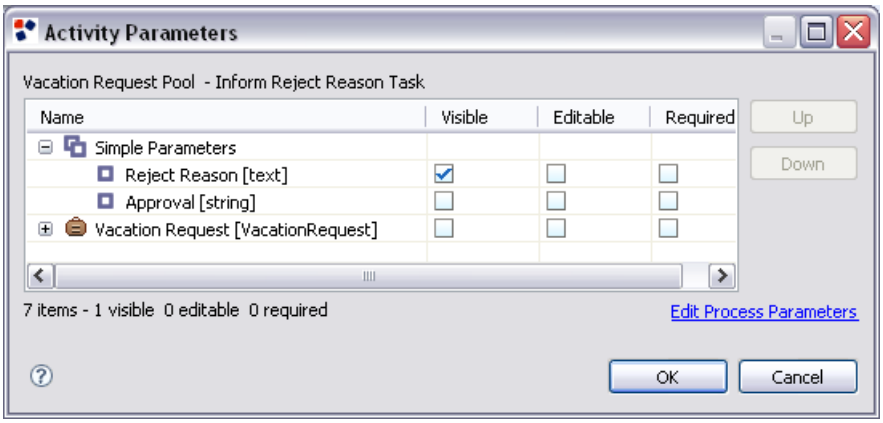

Figure 7.7: The parameters of the *Inform Reject Reason* activity

7.1.5 Events

- The *Start Event* will be triggered by an *Employee*, by clicking on a button.

- The process will reach the *End Event* either when the *Reject Reason* was informed or when the *Administrative Task* was finished.

7.1.6 Gateways

Verify Available Vacation Days

This Service Gateway evaluates the *Vacation Request* parameters and checks whether the vacation days are available or not (Figure 7.8).

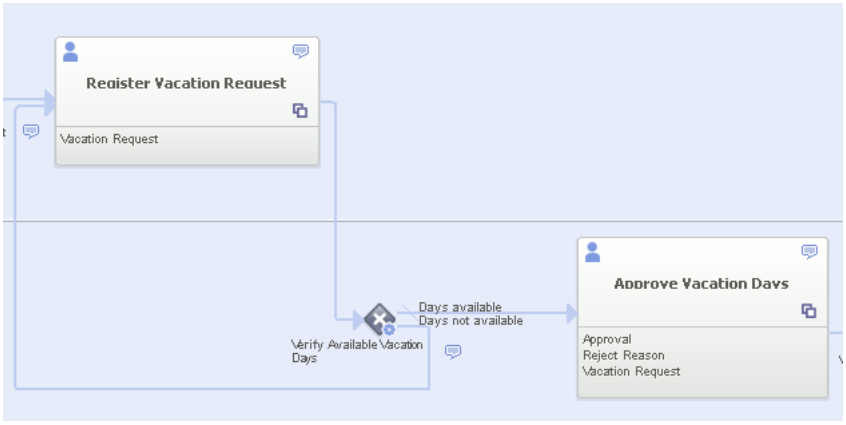

Figure 7.8: The *Verify Available Vacation Days* gateway

- If the vacation days are available the flow goes to the *Supervisor*. The next task that the supervisor may do is to *Approve Vacation Days*.
- If the days are not available the flow goes back to the *Employee*. The next task that the employee may do is to *Register Vacation Days* again.

This Service should be detailed at application level since it should connect to the *Company Information System* to define the vacation days availability.

As the expression's condition have to be evaluated when generating test data, this service is done as a simple comparison with the number of working days that the employee wants to take as holidays. The condition checks whether the number of days is superior or inferior to 30.

The configuration can be seen in the Figure 7.9 and 7.10.

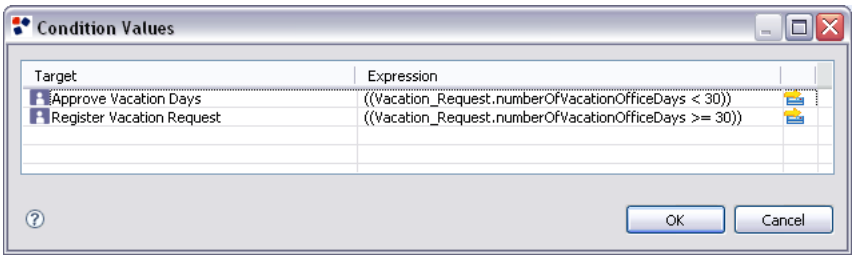

Figure 7.9: The gateway condition

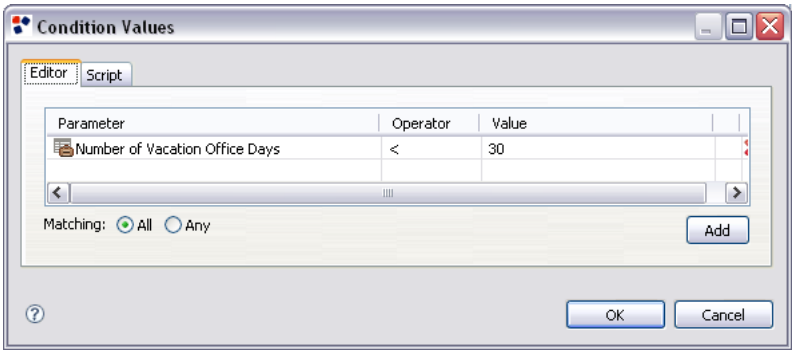

Figure 7.10: The configuration values for the gateway conditions

Verify Approval

This Service Gateway checks the supervisor's vacation days approval (Figure 7.11).

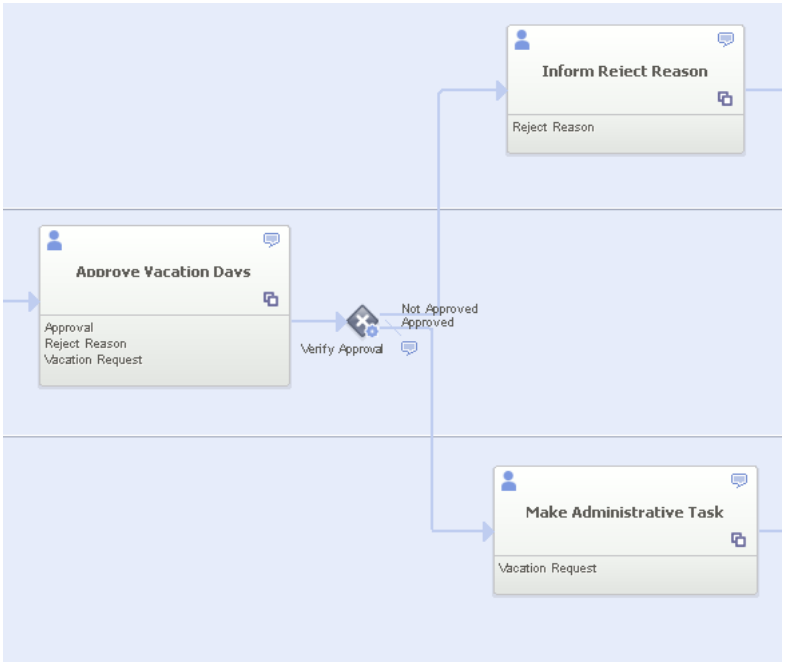

Figure 7.11: The *Verify Approval* gateway

- If the value is *Approved* (default option) the flow goes to the *Human Resource Assistant*. The next task that the assistant may do is to *Make Administrative Task*.
- If the value is *Not Approved* the flow goes back to the *Employee*. The next task that the employee may do is to see the reject reason.

The next figure shows how the different exit paths of this gateway are customized depending on the *Approved/Not Approved* values (assigned as seen in the Figure 7.3).

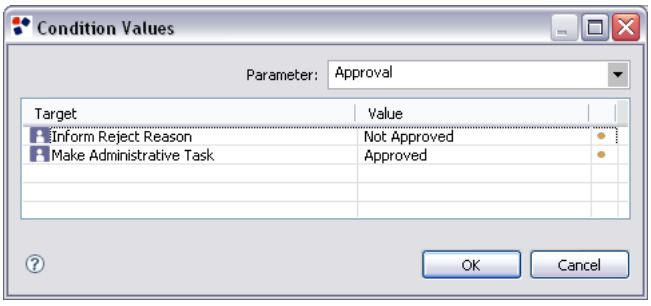

Figure 7.12: The *Condition Values* for the gateway

7.2 The exported XPDL from WebRatio

The exported XPDL file corresponding to the *Vacation Request* business process is shown in the Apendix 9.1. Remember that the shown XPDL corresponds to the completed version of the file that should be exported from *WebRatio*. This was explained in detail in the Section 6.2.1.

7.3 The generated output

After executing the implemented approach exposed in the Chapter 6, the corresponding generated output is produced and it is shown in the Appendix 9.2.

7.4 Running the example

To begin with, the implemented source code is organized inside the *src* package, in this way:

- *algorithms*: This package has the code corresponding to both proposed algorithms for the *Paths Generator* and *Test Data Generator*. Its therefore subdivided into two sub-packages: *data* and *paths*.
- *facade*: Contains the *Facade* class connecting all the exposed components. The calls done in this *Facade* class are being shown in the Figure 6.7.
- *generateOutput*: Contains the *Test Case Generator* implementation and the *Output File Parser* called *XMIGenerator*.
- *gui*: Contains a small *gui* interface to give as input the *XPDL* and shows the final *XML* file.
- *myGraph*: Contains the classes of the proposed *BPGraph*.
- *readInput*: Contains all the classes corresponding to the *Input File Parser* called *XPDLReader*.

The configuration files can be found in:

src : algorithms : paths : pathGenerator.properties
src : algorithms : data : testData.properties
src : algorithms : data : dataTypes : date.properties
src : algorithms : data : dataTypes : number.properties
src : algorithms : data : dataTypes : string.properties
src : algorithms : data : dataTypes : text.properties

Finally, to run the complete example of the implemented *Test Case Generator*, this steps can be followed:

1. Create the *Vacation Request* business process model using *WebRatio*.
   - This can be done using the version *BPM WebRatio 6.0.0 BETA* that can be downloaded free in *http://www.webratio.com*.
   - To create the example BPM Model, once you are in the workbench, follow *File : New : Example : Vacation Request*
   - To change the process parameters and its conditions you can follow the example images shown in this Section.
   - Export the *XPDL* selecting the *Vacation Request* process in the outline and following *Export : XML Process Definition Language 2.1*[1]
2. Run the class called *MyGui* and follow *File : Open* and choose the *XPDL* file that was extracted in the previous step. Afterwards, the *XPDL* will be shown in the screen. Choose the option *Generate : Test Cases*. Finally, the generated output file will be shown in the screen as shown in the Figure 7.13.

[1]Remember that the export into *XPDL* should be completed in order to obtain as result the same *Test Cases* shown in this Section. The expected *XPDL* is shown in the 9.2 and provided inside the source code packages containing the implementation

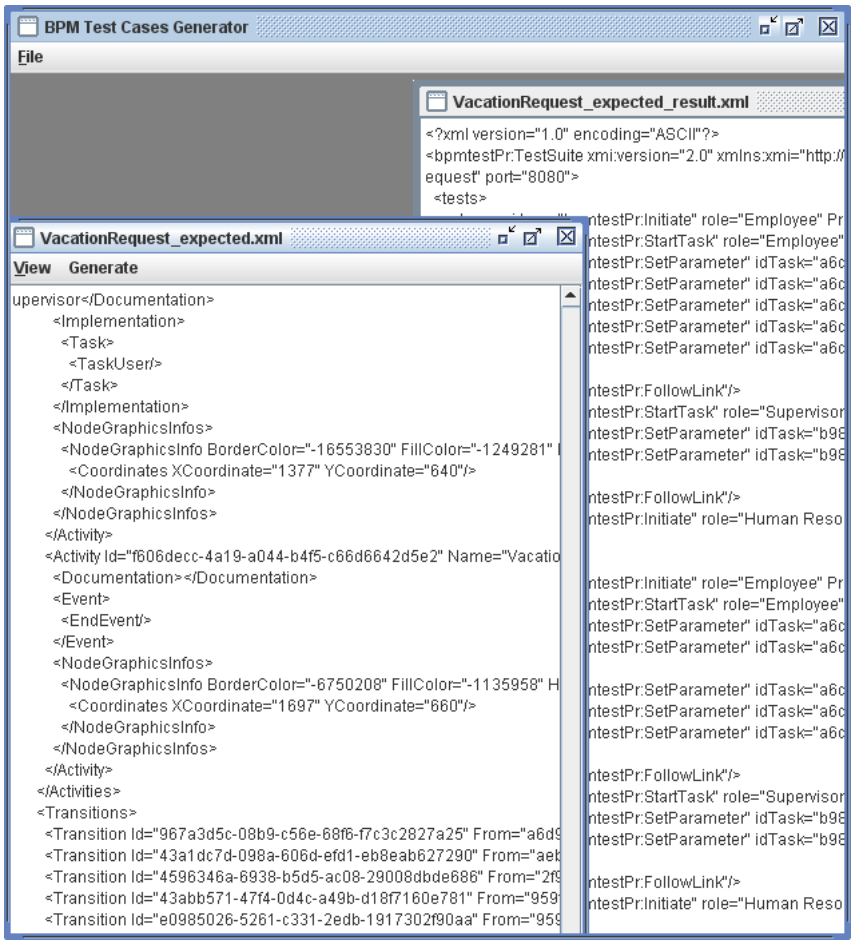

Figure 7.13: The *Graphical User Interface* used to run the example showing the *input* and *output* files

Chapter 8

Conclusions

The result of the present work is showing the feasibility of the automatic generation of a platform independent *Test Suite* given a *Business Process Model*.

One important advantage of the proposed design is to be flexible enough to be adapted and used in many different contexts. This is given because each of the constituting components is thought to be uncoupled and independent from the rest. In this way, many algorithms and approaches could be evaluated for each component in order to find the best combination possible for a given environment.

Finally, as shown with the implementation prototype, the possibility of having a platform independent *Test Case Generator* is not a hard task and it can be done in a fast and easy way using already existing algorithms.

8.1 Summary of contributions

The contributions of this work are summarized as follows:

- To be the first BPM based test case generation solution integrating paths exploration and test data generation.
- The possibility to configure the coverage depth of the resulting test cases.
- The interchangeability of each component. The proposed approach is not meant to work with just one *path exploration algorithm* or one *test data generation approach* but many different ones. This allows the tailoring for any specific need that any test case generator may require.
- The possibility of configuring each component differently in order to get different results even if the same algorithms are being used.

8.2 Future Work

The future work can be focused basically in the optimization of each of the components independently. For example:

- *Input File*: Within *WebRatio*, the adaptation of the XPDL exporter is needed in order to have the complete information coming from the business process model (Specified in the Section 6.2.1).
- *Paths exploration*:
  - Include the implementation algorithms needed to detect the different control constructs exposed in the section 5.3.2. This may need also to remove some possible nested constructs in order to simplify the model and the customization of the generation of test cases.
  - To make a possible classification of the explored paths (that will be translated into test cases). Therefore, each test case (path) could be classified as critical, semi-critical or default. Based on this classification, the execution of test cases may differ.
- *Test Data Generation*:
  - The automatization of the test data generation. There are many tools that provide this functionality. The idea is to evaluate them and automatize this step.
  - Correlate the generated test data with the flow of the process. The data may be changed according to all the steps visited. This concept is more known as *Symbolic execution*. It has been proposed by [Cla76], [How77], [Off91].

However, other proposals can also complement the present approach. For example:

- *Assertions*: With the current proposal it can be only checked if the business process produce no errors during its execution. There is no way to check the results of each step while the tests are being executed. Therefore, a system for assertion is proposed to cover this lack. Within the *WebRatio* tool, additional expressions should be inserted in order to complement the model and later be extracted by the assertion system.

Bibliography

[AHKB03] W. M. P Van Der Aalst, A. H. M Ter Hofstede, B. Kiepuszewski, and A. P Barros. Workflow patterns. *Distrib. Parallel Databases*, 14(1):551, 2003.

[BBF10] Marco Brambilla, Stefano Butti, and Piero Fraternali. WebRatio BPM: a tool for designing and deploying business processes on the web. In Boualem Benatallah, Fabio Casati, Gerti Kappel, and Gustavo Rossi, editors, *Web Engineering*, volume 6189, page 415429. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[BBM03] Frank Budinsky, Stephen A Brodsky, and Ed Merks. *Eclipse Modeling Framework*. Pearson Education, 2003.

[Bei03] Boris Beizer. *Software testing techniques*. Dreamtech, New Delhi, 2nd ed. edition, 2003.

[Bin00] Robert Binder. *Testing object-oriented systems : models, patterns, and tools*. Addison-Wesley, Reading Mass., 2000.

[BL02] Lionel Briand and Yvan Labiche. A UML-Based approach to system testing. *Software and Systems Modeling*, 1(1):10–42, 2002.

[BLL04] Xiaoqing Bai, C. Peng Lam, and Huaizhong Li. An approach to generate the thin-threads from the UML diagrams. In *Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004.*, pages 546–552, Hong Kong, 2004.

[Bud04] Frank Budinsky. *Eclipse modeling framework: a developer's guide*. Addison-Wesley, Boston, 2004.

[Cla76] L.A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, SE-2(3):215–222, 1976.

[CLLW05] Rob Chandler, Huaizhong Li, Chiou Peng Lam, and Mount Lawley Wa. *Generating Usage Scenarios Automatically from UML Activity Diagrams.* 2005.

[Dai04] Zhen Ru Dai. Model-Driven testing with UML 2.0. Technical report, Computing Laboratory, University of Kent, 2004.

[DtGF06] Trung T. Dinh-trong, Sudipto Ghosh, and Robert B. France. A systematic approach to generate inputs to test UML design models. In *In Proceedings of the 17th International Symposium on Software Reliability Engineering*, page 95104, 2006.

[FK96] Roger Ferguson and Bogdan Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology*, 5(1):63–86, 1996.

[FT10] Piero Fraternali and Massimo Tisi. Multi-level tests for model driven web applications. In Boualem Benatallah, Fabio Casati, Gerti Kappel, and Gustavo Rossi, editors, *Web Engineering*, volume 6189 of *Lecture Notes in Computer Science*, pages 158–172. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-13911-6_11.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[How75] W.E. Howden. Methodology for the generation of program test data. *IEEE Transactions on Computers*, C-24(5):554–560, 1975.

[How77] W.E. Howden. Symbolic testing and the DISSECT symbolic evaluation system. *IEEE Transactions on Software Engineering*, SE-3(4):266–278, 1977.

[Hun09] J Hunter. The JDOM project. http://www.jdom.org/, 2009.

[HVFR05] Jean Hartmann, Marlon Vieira, Herbert Foster, and Axel Ruder. A UML-based approach to system testing. *Innovations in Systems and Software Engineering*, 1:12–24, 2005. 10.1007/s11334-005-0006-0.

[JGr10] JGraphT. JGraphT API. http://www.jgrapht.org/javadoc/, 2010.

[JSS00] Daniel Jackson, Ian Schechter, and Hya Shlyahter. Alcoa, the alloy constraint analyzer. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, page 730733, New York, NY, USA, 2000. ACM.

[KHBC99] Y. G. Kim, H. S. Hong, D. H. Bae, and S. D. Cha. Test cases generation from UML state diagrams. *IEE Proceedings - Software*, 146(4):187, 1999.

[KKBK07] Hyungchoul Kim, Sungwon Kang, Jongmoon Baik, and Inyoung Ko. Test cases generation from UML activity diagrams. In *Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD 2007)*, pages 556–561, Qingdao, 2007.

[Kor90] B. Korel. Automated software test data generation. *IEEE Trans. Softw. Eng.*, 16(8):870879, 1990.

[Kor96] Bogdan Korel. Automated test data generation for programs with procedures. In *Proceedings of the 1996 international symposium on Software testing and analysis - ISSTA '96*, pages 209–215, San Diego, California, United States, 1996.

[Lar02] Craig Larman. *Applying UML and patterns: an introduction to object-oriented analysis and design and the unified process*. Prentice Hall PTR, Upper Saddle River NJ, 2nd ed. edition, 2002.

[LJX+04] Wang Linzhang, Yuan Jiesong, Yu Xiaofeng, Hu Jun, Li Xuandong, and Zheng Guoliang. Generating test cases from UML activity diagram based on Gray-Box method. In *11th Asia-Pacific Software Engineering Conference*, pages 284–291, Busan, Korea, 2004.

[MM03] J. Miller and J. Mukerji. MDA guide version 1.0.1. Technical report, Object Management Group (OMG), 2003.

[MMS01] C.C. Michael, G. McGraw, and M.A. Schatz. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, 27(12):1085–1110, December 2001.

[Mon10] Filippo Mondello. *Multilevel testing for model driven web applications*. PhD thesis, Politecnico di Milano, Como, 2010.

[MXX06] Chen Mingsong, Qiu Xiaokang, and Li Xuandong. Automatic test case generation for UML activity diagrams. In *Proceedings of the 2006 international workshop on Automation of software test - AST '06*, page 2, Shanghai, China, 2006.

[NFTJ03] C. Nebut, F. Fleurey, Y. Le Traon, and J. Jezequel. Requirements by contracts allow automated system testing. In *14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003.*, pages 85–96, Denver, Colorado, USA, 2003.

[NFTmJ03] Clmentine Nebut, Franck Fleurey, Yves Le Traon, and Jean marc Jzquel. A Requirement-Based approach to test product families. In *Proc. Fifth Workshop Product Families Eng*, page 198210. Springer Verlag, 2003.

[NS09] Ashalatha Nayak and Debasis Samanta. Synthesis of test scenarios using UML activity diagrams. *Software and Systems Modeling*, pages 1–27, 2009. 10.1007/s10270-009-0133-4.

[NS10] Ashalatha Nayak and Debasis Samanta. Automatic test data synthesis using UML sequence diagrams. *Journal of Object Technology*, 9(2):115–144, March 2010.

[Nta88] S.C. Ntafos. A comparison of some structural testing strategies. *IEEE Transactions on Software Engineering*, 14(6):868–874, 1988.

[OA99] Jeff Offutt and Aynur Abdurazik. Generating tests from UML specifications. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723, page 416429. Springer, 1999.

[oD00] Department of Defense. *End-to-End Integration Test Guidebook. Technical Report. Version 1.0.* 2000.

[Off91] A. Offutt. An integrated automatic test data generation system. *Journal of Systems Integration*, 1:391–409, 1991. 10.1007/BF02262722.

[OMG09a] OMG. *OMG Unified Modeling Language TM (OMG UML), Superstructure*. OMG, 2009.

[OMG09b] Object Management Group OMG, editor. *BPMN Business Process Modeling Notation*. January 2009.

[PAK+07] Orest Pilskalns, Anneliese Andrews, Andrew Knight, Sudipto Ghosh, and Robert France. Testing UML designs. *Inf. Softw. Technol.*, 49(8):892912, 2007.

[Rum03] Bernhard Rumpe. Model-Based testing of Object-Oriented systems. In Frank S de Boer, Marcello M Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, volume 2852 of *Lecture Notes in Computer Science*, pages 380–402. Springer Berlin / Heidelberg, 2003. 10.1007/978-3-540-39656-7_16.

[RW85] Sandra Rapps and Elaine J Weyuker. Selecting software test data using data flow information. *IEEE Trans. Softw. Eng.*, 11(4):367375, 1985.

[Sed83] Robert Sedgewick. *Algorithms*. Addison-Wesley, Reading Mass., 1983.

[Sha08] Robert M. Shapiro. XPDL 2.1 integrating process interchange and BPMN, 2008.

[Sim09] Ricardo Simoes. APAC an exact algorithm for retrieving cycles and paths in all kinds of graphs. *Tkhne - Revista de Estudos Politcnicos*, 2009.

[SMK07] Philip Samuel, Rajib Mall, and Pratyush Kanth. Automatic test case generation from UML communication diagrams. *Inf. Softw. Technol.*, 49(2):158171, 2007.

[Tay83] Richard N Taylor. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica*, 19:57–84, 1983. 10.1007/BF00263928.

[WS08] Stephan Weileder and Bernd-Holger Schlingloff. Deriving input partitions from UML models for automatic test generation. In Holger Giese, editor, *Models in Software Engineering*, volume 5002 of *Lecture Notes in Computer Science*, pages 151–163. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-69073-3_17.

[XLL05] Dong Xu, Huaizhong Li, and Chiou Peng Lam. Using adaptive agents to automatically generate test scenarios from the UML activity diagrams. In *12th Asia-Pacific Software Engineering Conference (APSEC'05)*, pages 385–392, Taipei, Taiwan, 2005.

Chapter 9

Appendices

9.1 The exported XPDL from WebRatio

Listing 9.1: Example of *Text* values properties file

```
<?xml version="1.0" encoding="UTF-8"?>
<Package xmlns="http://www.wfmc.org/2008/XPDL2.1">
  <PackageHeader>
    <XPDLVersion>2.1</XPDLVersion>
    <Vendor>WebRatio</Vendor>
    <Created>2010-08-14T10:52:59.656+0200</Created>
    <Description></Description>
    <Documentation>This is a sample WebRatio BPM Project. It shows a small business↩
        process. The process starts with the creation of an Vacation Request by an↩
        Employee and manages all the following approval steps.</Documentation>
  </PackageHeader>
  <RedefinableHeader>
    <Countrykey>US</Countrykey>
  </RedefinableHeader>
  <ExternalPackages/>
  <Participants/>
  <Pools>
    <Pool Id="5a7d680b-de07-5ac1-adfa-d69cb2c3cb24" Name="Vacation Request" ↩
        BoundaryVisible="true" Process="cc644979-c239-8212-deee-f3ec1bbe4a84">
      <Lanes>
        <Lane Id="dc08795b-bc6c-0e9e-8d2b-3a19c12ae294" Name="Employee" ParentPool=↩
            "5a7d680b-de07-5ac1-adfa-d69cb2c3cb24">
          <Documentation></Documentation>
          <NodeGraphicsInfos>
            <NodeGraphicsInfo BorderColor="-11513776" FillColor="-1315861" Height="↩
                250" Width="1427">
              <Coordinates XCoordinate="50" YCoordinate="0"/>
            </NodeGraphicsInfo>
          </NodeGraphicsInfos>
        </Lane>
        <Lane Id="7ab7de78-58ef-65c5-0eaf-c2fad1603450" Name="Supervisor" ↩
            ParentPool="5a7d680b-de07-5ac1-adfa-d69cb2c3cb24">
          <Documentation></Documentation>
          <NodeGraphicsInfos>
```

```
            <NodeGraphicsInfo BorderColor="-11513776" FillColor="-1315861" Height="↩
                240" Width="1180">
              <Coordinates XCoordinate="50" YCoordinate="250" />
            </NodeGraphicsInfo>
          </NodeGraphicsInfos>
        </Lane>
        <Lane Id="f9cd2b82-1f05-49ee-d282-454b1633ab8d" Name="Human Resource ↩
            Assistant" ParentPool="5a7d680b-de07-5ac1-adfa-d69cb2c3cb24">
          <Documentation></Documentation>
          <NodeGraphicsInfos>
            <NodeGraphicsInfo BorderColor="-11513776" FillColor="-1315861" Height="↩
                220" Width="1687">
              <Coordinates XCoordinate="50" YCoordinate="490" />
            </NodeGraphicsInfo>
          </NodeGraphicsInfos>
        </Lane>
      </Lanes>
      <NodeGraphicsInfos>
        <NodeGraphicsInfo BorderColor="-16777216" FillColor="-1" Height="710" Width↩
            ="1717">
          <Coordinates XCoordinate="30" YCoordinate="30" />
        </NodeGraphicsInfo>
      </NodeGraphicsInfos>
    </Pool>
  </Pools>
  <MessageFlows />
  <Associations />
  <Artifacts />
  <Parameters>
    <Parameter Name="Request Date" Type="date" />
    <Parameter Name="Employee Name" Type="string" />
    <Parameter Name="Start Date" Type="date" />
    <Parameter Name="End Date" Type="date" />
    <Parameter Name="Number of Vacation Office Days" Type="number" />
    <Parameter Name="Approval" Type="string" />
    <Parameter Name="Reject Reason" Type="text" />
  </Parameters>
  <WorkflowProcesses>
    <WorkflowProcess Id="cc644979-c239-8212-deee-f3ec1bbe4a84" Name="Vacation ↩
        Request">
      <ActivitySets />
      <Activities>
        <Activity Id="a6d9de86-838c-655c-5890-c2bd82408600" Name="Register Vacation↩
            Request" IdLane="dc08795b-bc6c-0e9e-8d2b-3a19c12ae294" ExecutionType="↩
            User">
          <Documentation>This Task allows the Employee to define all the details of↩
              the Vacation Request</Documentation>
          <Implementation>
            <Task>
              <TaskUser />
            </Task>
          </Implementation>
          <Parameters>
              <Parameter Name="Request Date" />
              <Parameter Name="Employee Name" />
              <Parameter Name="Start Date" />
              <Parameter Name="End Date" />
              <Parameter Name="Number of Vacation Office Days" />
          </Parameters>
          <NodeGraphicsInfos>
            <NodeGraphicsInfo BorderColor="-16553830" FillColor="-1249281" Height="↩
                60" Width="100">
```

```
            <Coordinates XCoordinate="318" YCoordinate="106"/>
          </NodeGraphicsInfo>
        </NodeGraphicsInfos>
      </Activity>
      <Activity Id="aeb2474c-52fc-a7c6-6b57-c257abbda2db" Name="New Vacation ↩
          Request" IdLane="dc08795b-bc6c-0e9e-8d2b-3a19c12ae294" ExecutionType="↩
          User">
        <Documentation>This Start Event will be triggered by an Employee, by ↩
            clicking on a button</Documentation>
        <Event>
          <StartEvent Trigger="None"/>
        </Event>
        <NodeGraphicsInfos>
          <NodeGraphicsInfo BorderColor="-10311914" FillColor="-1638505" Height="↩
              30" Width="30">
            <Coordinates XCoordinate="170" YCoordinate="130"/>
          </NodeGraphicsInfo>
        </NodeGraphicsInfos>
      </Activity>
      <Activity Id="2f92cd86-5912-403d-fc32-8a7cf70f28c3" Name="Inform Reject ↩
          Reason" IdLane="dc08795b-bc6c-0e9e-8d2b-3a19c12ae294" ExecutionType="↩
          User">
        <Documentation>If the Vacation Request has been rejected, this Task ↩
            informs the Employee of the Reject Reason</Documentation>
        <Implementation>
          <Task>
            <TaskUser/>
          </Task>
        </Implementation>
        <NodeGraphicsInfos>
          <NodeGraphicsInfo BorderColor="-16553830" FillColor="-1249281" Height="↩
              60" Width="100">
            <Coordinates XCoordinate="1367" YCoordinate="180"/>
          </NodeGraphicsInfo>
        </NodeGraphicsInfos>
      </Activity>
      <Activity Id="959f5c60-988d-176e-09e4-dead1e516855" Name="Verify Available ↩
          Vacation Days" IdLane="7ab7de78-58ef-65c5-0eaf-c2fad1603450" ↩
          ExecutionType="Service">
        <Documentation>This Service Gateway evaluates the Vacation Request ↩
            parameters and checks whether the vacation days are available or not.↩
             If the vacation days are available the flow goes to the Supervisor. ↩
            If the Days are not available the flow goes back to the Employee. ↩
            This Service must be detailed at application level since it should ↩
            connect to the Company Information System to define the vacation days↩
             availability</Documentation>
        <Route ConditionType="Expression">
            <Target TargetActivityId="b98a798f-11cc-7d4d-c985-624fa16c3e8e" ↩
                Matching="All">
              <Condition ParameterName="Number of Vacation Office Days" ↩
                  LogicOperator="lower" Value="30"/>
              <Condition ParameterName="Employee Name" LogicOperator="equals"↩
                  Value="Luz"/>
            </Target>
            <Target TargetActivityId="a6d9de86-838c-655c-5890-c2bd82408600" ↩
                Matching="All">
              <Condition ParameterName="Number of Vacation Office Days" ↩
                  LogicOperator="higherOrEquals" Value="30"/>
            </Target>
        </Route>
        <NodeGraphicsInfos>
```

```
            <NodeGraphicsInfo BorderColor="-5855715" FillColor="-52" Height="40" ↩
                Width="40">
              <Coordinates XCoordinate="607" YCoordinate="430" />
            </NodeGraphicsInfo>
          </NodeGraphicsInfos>
        </Activity>
        <Activity Id="b98a798f-11cc-7d4d-c985-624fa16c3e8e" Name="Approve Vacation ↩
            Days" IdLane="7ab7de78-58ef-65c5-0eaf-c2fad1603450" ExecutionType="User↩
            ">
          <Documentation>This Task allows the Supervisor to decide to approve or ↩
              not the Vacation Request</Documentation>
          <Parameters>
            <Parameter Name="Approval" />
            <Parameter Name="Reject Reason" />
          </Parameters>
          <Implementation>
            <Task>
              <TaskUser />
            </Task>
          </Implementation>

          <NodeGraphicsInfos>
            <NodeGraphicsInfo BorderColor="-16553830" FillColor="-1249281" Height="↩
                60" Width="100">
              <Coordinates XCoordinate="940" YCoordinate="410" />
            </NodeGraphicsInfo>
          </NodeGraphicsInfos>
        </Activity>
        <Activity Id="9dacd19b-21b3-3e84-6fe3-60baaa0f3cb6" Name="Verify Approval" ↩
            IdLane="7ab7de78-58ef-65c5-0eaf-c2fad1603450" ExecutionType="Service">
          <Documentation>This Service Gateway checks the Supervisor Vacation Days ↩
              approval. If the value is "Approved" (default option) the flow goes ↩
              to the Human Resource Assistant. If the value is "Not Approved" the ↩
              flow goes back to the Employee</Documentation>
          <Route ConditionType="Literal">
                <Target TargetActivityId="19f0d799-60c9-d758-9735-7ca02a809ea8">
                      <Condition ParameterName="Approval" Value="Approved" />
                </Target>
                <Target TargetActivityId="2f92cd86-5912-403d-fc32-8a7cf70f28c3">
                        <Condition ParameterName="Approval" Value="Not Approved" />
                </Target>
          </Route>
          <NodeGraphicsInfos>
            <NodeGraphicsInfo BorderColor="-5855715" FillColor="-52" Height="40" ↩
                Width="40">
              <Coordinates XCoordinate="1180" YCoordinate="440" />
            </NodeGraphicsInfo>
          </NodeGraphicsInfos>
        </Activity>
        <Activity Id="19f0d799-60c9-d758-9735-7ca02a809ea8" Name="Make ↩
            Administrative Task" IdLane="f9cd2b82-1f05-49ee-d282-454b1633ab8d" ↩
            ExecutionType="User">
          <Documentation>This Task allows the Human Resource Assistant to perform ↩
              the administrative task in order to register the Vacation Request if ↩
              it has been approved by the Supervisor</Documentation>
          <Implementation>
            <Task>
              <TaskUser />
            </Task>
          </Implementation>
          <NodeGraphicsInfos>
```

```
            <NodeGraphicsInfo BorderColor="-16553830" FillColor="-1249281" Height="↩
                60" Width="100">
              <Coordinates XCoordinate="1377" YCoordinate="640" />
            </NodeGraphicsInfo>
          </NodeGraphicsInfos>
        </Activity>
        <Activity Id="f606decc-4a19-a044-b4f5-c66d6642d5e2" Name="Vacation Request ↩
            End" IdLane="f9cd2b82-1f05-49ee-d282-454b1633ab8d">
          <Documentation></Documentation>
          <Event>
            <EndEvent />
          </Event>
          <NodeGraphicsInfos>
            <NodeGraphicsInfo BorderColor="-6750208" FillColor="-1135958" Height="↩
                30" Width="30">
              <Coordinates XCoordinate="1697" YCoordinate="660" />
            </NodeGraphicsInfo>
          </NodeGraphicsInfos>
        </Activity>
      </Activities>
      <Transitions>
        <Transition Id="967a3d5c-08b9-c56e-68f6-f7c3c2827a25" From="a6d9de86-838c↩
            -655c-5890-c2bd82408600" To="959f5c60-988d-176e-09e4-dead1e516855" />
        <Transition Id="43a1dc7d-098a-606d-efd1-eb8eab627290" From="aeb2474c-52fc-↩
            a7c6-6b57-c257abbda2db" To="a6d9de86-838c-655c-5890-c2bd82408600" />
        <Transition Id="4596346a-6938-b5d5-ac08-29008dbde686" From="2f92cd86↩
            -5912-403d-fc32-8a7cf70f28c3" To="f606decc-4a19-a044-b4f5-c66d6642d5e2"↩
            />
        <Transition Id="43abb571-47f4-0d4c-a49b-d18f7160e781" From="959f5c60-988d↩
            -176e-09e4-dead1e516855" To="b98a798f-11cc-7d4d-c985-624fa16c3e8e" />
        <Transition Id="e0985026-5261-c331-2edb-1917302f90aa" From="959f5c60-988d↩
            -176e-09e4-dead1e516855" To="a6d9de86-838c-655c-5890-c2bd82408600" />
        <Transition Id="9f083977-e4bf-6b5f-2e6f-c13412cbf87d" From="b98a798f-11cc-7↩
            d4d-c985-624fa16c3e8e" To="9dacd19b-21b3-3e84-6fe3-60baaa0f3cb6" />
        <Transition Id="ab96bdbe-f436-73a4-2b0e-7f5cd4df27f0" From="9dacd19b-21b3-3↩
            e84-6fe3-60baaa0f3cb6" To="19f0d799-60c9-d758-9735-7ca02a809ea8" />
        <Transition Id="2a6f247a-5ae8-b221-031c-fdb58af673e0" From="9dacd19b-21b3-3↩
            e84-6fe3-60baaa0f3cb6" To="2f92cd86-5912-403d-fc32-8a7cf70f28c3" />
        <Transition Id="8e2ac007-7028-a2c3-92ce-198834bc2d3a" From="19f0d799-60c9-↩
            d758-9735-7ca02a809ea8" To="f606decc-4a19-a044-b4f5-c66d6642d5e2" />
      </Transitions>
    </WorkflowProcess>
  </WorkflowProcesses>
</Package>
```

9.2 The generated output

Listing 9.2: Example of *Text* values properties file

```
<?xml version="1.0" encoding="ASCII"?>
<bpmtestPr:TestSuite xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi↩
    ="http://www.w3.org/2001/XMLSchema-instance" xmlns:bpmtestPr="bpmtestUri" name=↩
    "localhost" basepath="Vacation Request" port="8080">
  <tests>
    <steps xsi:type="bpmtestPr:Initiate" role="Employee" ProcessName="New Vacation ↩
        Request" />
```

```
    <steps xsi:type="bpmtestPr:StartTask" role="Employee"/>
    <steps xsi:type="bpmtestPr:SetParameter" idTask="a6d9de86-838c-655c-5890-↩
        c2bd82408600" subprocessName="Register Vacation Request" value="01/02/03" ↩
        parameterName="Request Date"/>
    <steps xsi:type="bpmtestPr:SetParameter" idTask="a6d9de86-838c-655c-5890-↩
        c2bd82408600" subprocessName="Register Vacation Request" value="String ↩
        example" parameterName="Employee Name"/>
    <steps xsi:type="bpmtestPr:SetParameter" idTask="a6d9de86-838c-655c-5890-↩
        c2bd82408600" subprocessName="Register Vacation Request" value="01/02/03" ↩
        parameterName="Start Date"/>
    <steps xsi:type="bpmtestPr:SetParameter" idTask="a6d9de86-838c-655c-5890-↩
        c2bd82408600" subprocessName="Register Vacation Request" value="01/02/03" ↩
        parameterName="End Date"/>
    <steps xsi:type="bpmtestPr:SetParameter" idTask="a6d9de86-838c-655c-5890-↩
        c2bd82408600" subprocessName="Register Vacation Request" value="7" ↩
        parameterName="Number of Vacation Office Days"/>
    <steps xsi:type="bpmtestPr:FollowLink"/>
    <steps xsi:type="bpmtestPr:StartTask" role="Supervisor"/>
    <steps xsi:type="bpmtestPr:SetParameter" idTask="b98a798f-11cc-7d4d-c985-624↩
        fa16c3e8e" subprocessName="Approve Vacation Days" value="Not Approved" ↩
        parameterName="Approval"/>
    <steps xsi:type="bpmtestPr:SetParameter" idTask="b98a798f-11cc-7d4d-c985-624↩
        fa16c3e8e" subprocessName="Approve Vacation Days" value="this is an example↩
         of long texts" parameterName="Reject Reason"/>
    <steps xsi:type="bpmtestPr:FollowLink"/>
    <steps xsi:type="bpmtestPr:Initiate" role="Human Resource Assistant" ↩
        ProcessName="Vacation Request End"/>
  </tests>
  <tests>
    <steps xsi:type="bpmtestPr:Initiate" role="Employee" ProcessName="New Vacation ↩
        Request"/>
    <steps xsi:type="bpmtestPr:StartTask" role="Employee"/>
    <steps xsi:type="bpmtestPr:SetParameter" idTask="a6d9de86-838c-655c-5890-↩
        c2bd82408600" subprocessName="Register Vacation Request" value="01/02/03" ↩
        parameterName="Request Date"/>
    <steps xsi:type="bpmtestPr:SetParameter" idTask="a6d9de86-838c-655c-5890-↩
        c2bd82408600" subprocessName="Register Vacation Request" value="Value of a ↩
        String" parameterName="Employee Name"/>
    <steps xsi:type="bpmtestPr:SetParameter" idTask="a6d9de86-838c-655c-5890-↩
        c2bd82408600" subprocessName="Register Vacation Request" value="02/02/02" ↩
        parameterName="Start Date"/>
    <steps xsi:type="bpmtestPr:SetParameter" idTask="a6d9de86-838c-655c-5890-↩
        c2bd82408600" subprocessName="Register Vacation Request" value="01/04/02" ↩
        parameterName="End Date"/>
    <steps xsi:type="bpmtestPr:SetParameter" idTask="a6d9de86-838c-655c-5890-↩
        c2bd82408600" subprocessName="Register Vacation Request" value="2" ↩
        parameterName="Number of Vacation Office Days"/>
    <steps xsi:type="bpmtestPr:FollowLink"/>
    <steps xsi:type="bpmtestPr:StartTask" role="Supervisor"/>
    <steps xsi:type="bpmtestPr:SetParameter" idTask="b98a798f-11cc-7d4d-c985-624↩
        fa16c3e8e" subprocessName="Approve Vacation Days" value="Approved" ↩
        parameterName="Approval"/>
    <steps xsi:type="bpmtestPr:SetParameter" idTask="b98a798f-11cc-7d4d-c985-624↩
        fa16c3e8e" subprocessName="Approve Vacation Days" value="this is another ↩
        description" parameterName="Reject Reason"/>
    <steps xsi:type="bpmtestPr:FollowLink"/>
    <steps xsi:type="bpmtestPr:Initiate" role="Human Resource Assistant" ↩
        ProcessName="Vacation Request End"/>
  </tests>
</bpmtestPr:TestSuite>
```