On the usage of OWL for Business Process Simulation

JBPS: The Java Business Process Simulator

Damián Soriano

Advisor: Prof. Marco Colombetti

School of Industrial and Information Engineering

Politecnico di Milano

October 2013

Abstract

Business Process Modeling is being extensively used in mature companies that want to define and improve their way of working. Among the many efforts, the Business Process Management and Notation is becoming the standard for Business Process Modeling in a graphical and intuitive way, accurate for both business process analyst and technical users.

The Java Business Process Simulator is an attempt to promote the use of Business Process Management and Notation and to provide a software support for the implementation of such processes within the companies. The Java Business Process Simulator seeks to reduce the existing gap between the business process definition and the business process implementation.

The Java Business Process Simulator make extensively use of Ontologies, and in particular of the Web Ontology Language, to provide a common language for the definition of the business process vocabulary and constrains. Ontologies are integrated with the business process definition to provide a simulation of those processes in a full running application that can be used for testing requirements assumptions or for running it in production environments.

Sommario

Il Business Process Modeling viene ampiamente utilizzato nelle aziende mature che vogliono definire e migliorare il loro modo di lavorare. Tra le tante iniziative, il Business Process Management and Notation sta diventando lo standard per la definizione dei processi aziendali in un modo grafico e intuitivo, sia per i business analyst, sia per gli utenti tecnici.

Il Java Business Process Simulator è un tentativo di promuovere l'uso del Business Process Management and Notation e per fornire un supporto software per l'attuazione di tali processi all'interno delle aziende. Il Java Business Process Simulator cerca di ridurre il divario esistente tra la definizione dei processi e la sua implementazione.

Il Java Business Process Simulator fa uso esteso di ontologie, e in particolare del Web Ontology Language, per fornire un linguaggio comune per la definizione del vocabolario dei processi e dei suoi vincoli. Le ontologie sono integrate con la definizione dei processi aziendali per fornire una simulazione di questi processi in un'applicazione che puo essere utilizzata per testare le ipotesi dei requisiti o per l'esecuzione in un ambiente di produzione.

Acknowledgements

To my wife Mercedes for being always by my side, supporting and encouraging me always to persecute the impossible.

To my mother and my father for always teaching me the benefits of education.

To my siblings Iván, Diego, Romina and Agustín, for teaching me me that a world without sharing is a world not worth living.

To my director Marco Colombetti for being always more than patient with me.

Contents

List of Figures

Chapter 1

Introduction

The need for analysis and improvement of business process in companies had lead to the search of a definition of a common notation that allow engineers to define such process. From this need, the Business Process Management and Notation (BPMN) was created. The BPMN is graphical and standard notation, developed by the Object Management Group, to describe business process and is now becoming the standard regarding Process Engineering.

More and more applications are build to support engineers in the definition of such process. As an example, the Bizagi BPM Suite [1] is a Business Process Management solution that allow organizations to automate processes.

Separately, thanks to the constant attempts to develop the Semantic Web, as the vision of the future of the Web, ontologies frameworks and specifications have gain a big momentum in the last years. Ontologies are becoming a synonym of standardization and more and more companies are adopting them to describe some parts of their business models. This can be seen in the maturity and diversity of Ontology related applications and frameworks such as Apache Jena, Protégé and the Pellet Reasoner among others.

The OWL Web Ontology Language is intended to be used when the ontology needs to be processed by applications. OWL is part of the growing stack of W3C recommendations related to the Semantic Web, it is build upon XML, XML Schema, RDF and RDF Schema.

This two unrelated standardization are the motivations behind the Java Business Process Simulator. Using the Bizagi BPM Suite as inspiration, the JBPS receives as input the definition of the business process in the BPMN and the definition of the business models in OWL. From this it generates a simulation through the BPMN definition, asking the user for input when necessary and ensuring the underlying business model remains consistent thought all the diagram traversal.

We developed the JBPS prototype to show some key features we think are interesting in using BPMN and OWL in real business. The JBPS is not only intended for simulation or testing that the definition of the BPMN diagrams and the OWL ontology are well defined with respect to the user intention but it is also intended to be used as a production application. We expect that the JBPS can be used as the final production version of companies that want to specify it's internal process via BPMN and OWL. In this sense we see JBPS as being the supporting tool for companies that want to have a full running application from their process definition and the business modeling without any coding.

In this work we will start talking about some background knowledge that should be understood first, in particular how ontologies are defined and how to define business process using the BPMN primitives. We will then talk about the requirements of the JBPS and some design and implementation details that we found interesting in the development of the prototype. The most relevant Third-Party Software Components are presented afterwards, some of them dramatically influence the code of the JBPS. Finally we will present some study cases on how to use the JBPS and how it behaves with different process and ontologies. The Java Business Process Simulator is completely Open Source and can be found for downloading or forking in [2].

Chapter 2

Background

In this section we will describe some of the concepts and frameworks that should be first comprehend in order to understand the implications and use of this work. There are two main parts that we are combining in this work.

On one hand we have the use of formal systems in order to express the world situation regarding the business environment, in particular we are using Description Logic and some of it's implementation. On the other hand we make extensive use of graphical representations tools to describe the business workflow and this is achieved via the Business Process Model and Notation as we will explain.

2.1 Description Logic and OWL

A formal language is a precisely defined set of strings of symbols. This strings are formed by elements in an alphabet that can be combined in a precise way specified by the language. Formal languages are usually provided with a semantic function that makes a mapping between the strings in the languages and a know and stablished system, like the set theory. This function is gives meaning to the symbols found in the languages that otherwise would be meaningless sequences of tokens.

Among the many formal languages used in mathematics, one is of particular interest: Propositional Logic. Propositional Logic contains formulas that are interpreted as propositions. There are also inference rules and axioms that can be used to derive new formulas from others. An interesting characteristic of Propositional Logic is that it is decidable, meaning that given a well formed term, there is an automatic way of determining if that statement is true or false.

Another interesting formal language is First-Order Logic. First-Order Logic is more powerful than Propositional Logic in the sense that in can capture more complex entities. For example, Natural Numbers can be represented in First-Order Logic but cannot be represented in Propositional Logic. The major drawback of First-Order Logic is that it is semi-decidable, meaning that if a predicate is true, we can derive a proof in a finite number of steps, but if a formula is not true we may not decide in a finite number of steps that the formula is false in an automatic way.

Semi-decidability is acceptable if we are only interesting in the proof, but in Knowledge Systems, where we do not know in advance the truth value of a formula, is unacceptable. On the other hand, Propositional Logic is very weak to express interesting world facts even if it enjoys the benefit of decidability.

Description Logic is a knowledge representation formal language that lay between the expressive power of First-Order Logic and Propositional Logic. It can express more interesting facts than Propositional Logic but it cannot express the facts about Natural Numbers, as First-Order Logic can. The important part is that Description Logic is decidable and it fits our need of a decidable language that can be used as a knowledge representation language.

2.1.1 Description Language and *SROIQ(D)*

Description Logic, being a family of formal knowledge representation language, provides different choices one of them called *SROIQ(D)*. In *SROIQ(D)* there are classes or concepts that are somehow sets of individuals. Individuals are anything from the real world that can be represented, such as a table, a person or even an abstract entity. There are also relations or roles that are sets of pairs of individuals, like mathematical relations.

There are two special kind of classes that have a particular meaning:

- $\top$: Called *top* or universe, represents the set of all the individuals present in the discourse universe.
- $\bot$: Called *bottom* or empty, that represents the set that do not contain any individual in the universe, the empty set.

Given two classes C and D, a relation R, an individual a and a natural number n there are a number of ways we can combine this elements to form new kind of classes that can be used to express new kind of concepts.

- $\sim C$: All the elements of the universe that do not belong to the C class.
- $C \sqcap D$: All the elements of the universe that belong both to the class C and the class D.
- $C \sqcup D$: All the elements of the universe that belong to the class C or the class D.

- $\exists R.C$: Called *Existential Restriction*, represents all the elements x of the universe such as there exists an element y of the class C where the pair (x, y) belongs to R.
- $\forall R.C$: Called *Universal Restriction*, represents all the elements x of the universe such as if there exists an element y where the pair (x, y) belongs to R then the element y belongs to class C.
- $\geq n\ R.C$: Called *Min-Cardinality Restriction* represent all the elements x of the universe such as the cardinality of the set of all pairs (x, y) where y belongs to C is grater or equal than n.
- $\leq n\ R.C$: Called *Max-Cardinality Restriction* represent all the elements x of the universe such as the cardinality of the set of all pairs (x, y) where y belongs to C is less or equal than n.
- $\{a\}$: Represent simply the class of the individual a.

There is also a way to compose two give relations R and S in a way to create a new complex relation, this is called relation composition and is similar to the concept of functional composition in mathematics.

- $R \circ S$: Called *Relation Composition* is a new relation that contains pair of the form (x, z) if there exists an individual y in the universe such as that the pair (x, y) belongs to R and (y, z) belongs to S.

Given all this ways of creating new compelx classes or relations we may now want to make some statements about this entities. This statements will describe the facts of the world, it's assertions. There are three different places to state this assertions, the *TBox* where we describe facts about classes, *ABox* where we describe facts about individuals and *RBox* where we describe facts about relations.

With any two classes C and D a general concept or class inclusion represent the inclusion of class D by class C, meaning that every individual that belongs to class C also belongs to class D. All statements in the TBox are of this kind.

$$C \sqsubseteq D$$

For the ABox we can make assertion on the fact that some individual a are part of a given class C or we can also express the fact that a specific pair of individuals (x, y) is contained in a given relation R. They are called *Concept Assertion* and *Role Assertion* respectively and are represented as follows:

$$a \in C$$

$$(x, y) \in R$$

We can also use the RBox to express some conditions about the relations that we use in the modeling of the facts of the world. First of all we can express the fact that a relation R is a sub-relation of another relation S, meaning that all pairs that belongs to R also belongs to S.

$$R \dot{\sqsubseteq} S$$

We can also state that two relations R and S are equivalent, meaning that the both contains the same pairs of individuals, in other words, the underlying sets of pairs are equivalent.

$$R \doteq S$$

Finally we can express that two relations R and S are disjoint, this means that for all individual x of the universe there is no other individual y such that (x, y) belongs to R and (x, y) belongs to S. In other words, two disjoint properties cannot connect one x individual with the same individual y. This is expressed in the following way:

$$DisProp(R, S)$$

There are a number of restrictions that apply to the RBox regarding the relation composition. All the statements that we just describe for the RBox apply perfectly for simple relations R and S, that means that they are not relation compositions of other more primitive relations. When we introduce the composition there are some constrains that apply to the RBox. In general, this kind of constrains exists because of three possible reasons:

1. If we do not have the constrain we loss decidability.
2. Or if we do not loss decidability, the complexity became too high to be usable in real software.
3. Or it is not clear what happens about decidability, if we still have it or not.

2.1.2 Web Ontology Language

The Web Ontology Language (OWL) is a family for knowledge representation endorsed by the World Wide Web Consortium[1]. Some of the OWL variations have their semantics based on Description Logics, this is mainly because the decidability of Description Logic and the fact that knowledge derivation should be present in OWL.

The different OWL variants are the following:

[1]http://www.w3.org/

- OWL Lite: Primary intended to support hierarchy classification and simple constrains. It was mainly aimed to provide a simple tool support.
- OWL DL: Named due to the correspondence with Description Logic, was originally designed to provide the maximum expressiveness while maintaining decidability. Includes all the OWL language constructs, but also applies some restrictions to maintain decidability.
- OWL Full: Provides a more powerful expressiveness but drop decidability. In OWL Full for example a class can be treated as a collection of individuals or as an individual itself. The semantic of OWL Full is completely different from the other two OWL flavors.

Open World Assumption

The languages in the OWL family are based on the Open World Assumption. The Open World Assumption states that the truth value of a statement is independent of the knowledge of a single observer to be true. Practically if we cannot prove that a statement is true, we cannot conclude that the statement is false. It is the opposite concept of the closed world assumption, which states that any statement that is not known to be true is false.

In most knowledge representation systems, the Open World Assumption is used to model the fact that no single agent has complete knowledge of the universe. Let's see an example that illustrate the difference between the Close World Assumption and the Open World Assumption.

Statement: Knowledge Engineering is a course at Politecnico di Milano.

Question: Is Artificial Intelligence a course at Politecnico di Milano.

Closed World answer: No, it is not.

Open World answer: I don't know.

From this example, it should be clear that the Open World Assumption is the natural assumption to be done in a knowledge representation system.

2.2 Business Processes

In this section we will describe the basics of the Business Process Modeling and the Business Process Model and Notation.

2.2.1 Background

In order to understand the idea and novelty of the *Business Process Model and Notation* it is first necessary to understand some basic concepts related to the process engineering.

The very first concept we should start describing is that of a *business process*. A *business process* is a collection of linked and structured tasks that results in a production of a specific service or product within a company.

There are different kind of business process, such as:

- *Operational Processes* that carry the core business of a company such as the Purchase or Sales processes.
- *Management Processes* that describe the operation of a system.
- *Supporting Processes* that are processes that supports the *Operational Processes* such as Accounting or Recruitment.

Each *process* can be broken down in different *sub-process* that at the very bottom are described in *tasks*. Each task is an activity that should be accomplished in a specific time period and is precisely defined. Tasks are then grouped, sometimes in dependency relations, in order to produce different processes.

Business Process Modeling is the activity of representing business process in a clear and standard notation in order to analyze and improve the underlying process. The final goal of the Business Process Modeling activity is to increased effectiveness and increased efficiency of the analyzed processes. Analyzed processes can be processes that already exists within the company or can be new processes that the company is trying to define in order to incorporate in it's inner functionality.

2.2.2 The Business Process Model and Notation

The *Business Process Model and Notation* is a standard graphical representation to describe business process. It was the developed by the *Object Management Group* (OMG) and it's current specification, version 2.0, can be found in [4].

The OMG is an open membership, not profitable computer centric industry standard consortium that produces and maintains computer industry specifications for interoperable, portable and reusable enterprise applications in distributed, heterogeneous environments. Membership includes Information Technology vendors, end users, government agencies and academia.

The scope of the BPMN is to provide an intuitive description of the business process for both technical and business users in order to create a standardized bridge to overcome the gap between business processes design and business processes implementation. The aim is to create a graphical language that process modelers can recognize and understand. The BPMN specification enables the standardization and portability or *process* definition, so it can be used and transfered to different vendor's environments.

In order to handle the complexity of the business processes and yet provide a simple and understandable graphical mechanism to describe them, the BPMN graphical elements are organized in categories. The five categories of elements are:

1. Flow Objects
2. Data
3. Connecting Objects
4. Swim-lanes
5. Artifacts

Flow Objects are the main describing elements in the BPMN diagrams and of three kinds: *Events*, *Activities* and *Gateways*.

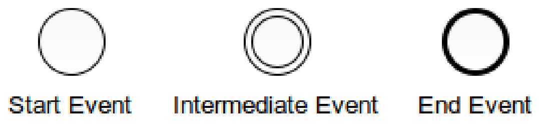


Figure 2.1: BPMN Events

- *Events*: Events are something that happens during the execution of a process. Events affect the flow of the model and have causes or impacts. Events are shown in Figure 2.1. There are three kind of Events, based on when they affect the flow.
  - *Start event*
  - *Intermediate event*
  - *End event*

- *Activities*: Activities represent generic work that companies do in their processes. There are different kind of activities, like sub-process or tasks. Tasks represent a single unit of work that cannot be broken down and there are different kind of tasks as shown in Figure 2.2.

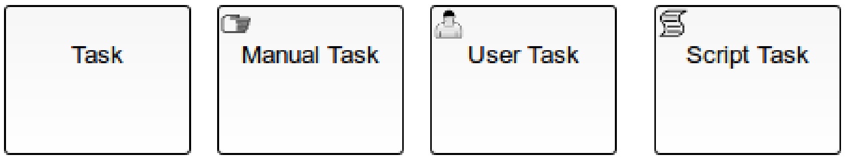


Figure 2.2: BPMN Activities

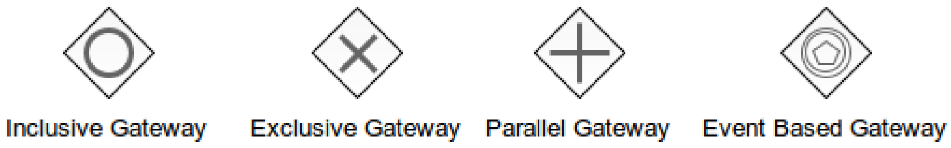


Figure 2.3: BPMN Gateways

- *Gateways*: A Gateway is used to control the divergence and convergence of Sequence Flows in a Process, they can be used for branching, forking, merging and joining paths, some of them shown in Figure 2.3. The type of Gateways are the following:
  - *Exclusive Gateways*
  - *Event Based Gateways*
  - *Parallel Gateways*
  - *Inclusive Gateways*
  - *Exclusive Event Based Gateways*
  - *Complex Gateways*
  - *Parallel Event Based Gateways*

There are different ways of connecting *Flow Objects* using *Connecting Objects* as shown in Figure 2.4.

- *Sequence Flow*: This can of connection are used to show the order that activities are performed in a business process.
- *Message Flow*: Message Flows are used to indicate the presence of a Message between two participants in the diagram.
- *Association*: Associations are used to link information and Artifacts with BPMN graphical elements in the BPMN diagrams.

Swim-lanes are a way of grouping and categorize visual BPMN elements in the diagrams. There are two type of Swim-lanes:

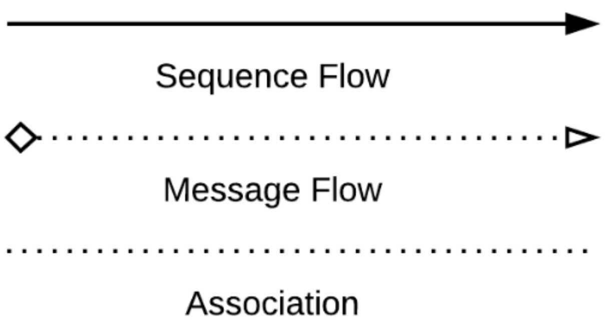


Figure 2.4: BPMN Connections

Figure 2.5: BPMN Pools

- *Pools*: Are a visual graphical representation of Participants in the business process as shown in Figure 2.5. A pool can contain one or more lanes. Pools can be open, meaning that internal details are shown in the diagram, or it can be a black box pool, where the internal details are hidden.

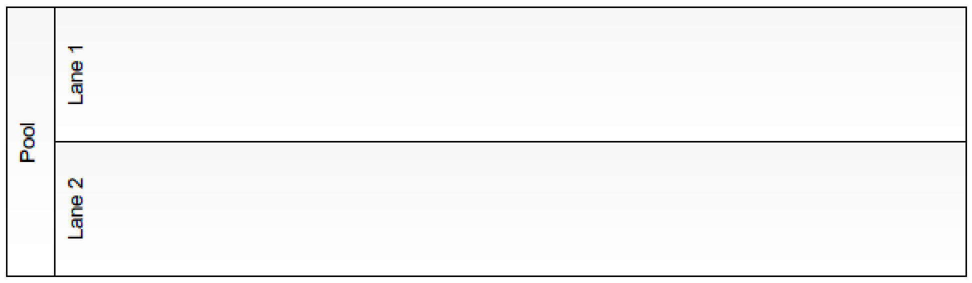


Figure 2.6: BPMN Lanes

- *Lanes*: A lane is a sub-partition in a process, sometimes within a pool, as shown in Figure 2.6. They do extend the entire length of the Process and are usually used to organize and categorize Activities within a BPMN diagram.

2.2.3 Examples

In this section we will describe the *Pizza Business Process* firs described in [3] and then expanded in [5] to illustrate some of the concepts of the BPMN elements and diagrams. The process will describe from the customer selecting and ordering a pizza by calling the restaurant, until the pizza is delivered to the customer.

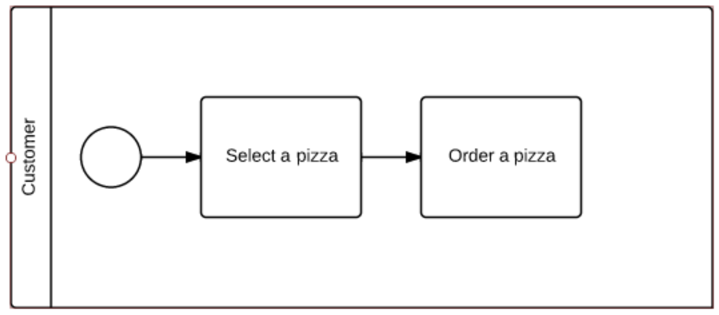


Figure 2.7: The Pizza Process: Customer Lane

The whole process start with a customer willing to eat some pizza, this would be the starting point of our BPMN diagram, the customer willing to order some pizza will be the start event as we can see in Figure 2.7. After the starting event the customer should first select which pizza he want to eat and then order it. Each action the user should take is represented with a task in the diagram and the order between each task and the event is described by the Sequence Flow objects.

After the customer order a pizza a new swim-lane appears that capture the pizza making process of the restaurant, since in this process there are more than one person involved, the swim-lane will be a poll with three lanes, one for each of the participants: the Clerk, the Pizza Chef and the Delivery Boy.

When the Customer order a pizza the Clerk receive the order, meaning that a start message event begins in the pizza vendors, this event is represented in Figure 2.8 with the envelope in the single-lined circle. The dashed line indicates that the message is passing by and the arrow indicates the direction of the message.

After the Clerk took the order he will pass it to the Pizza Chef, as shown in the diagram by the solid line that end up in the Pizza Chef's lane, in the *Bake the pizza* task. After the Pizza Chef finish baking the pizza he will pass it to the Delivery Boy as can be seen in the diagram.

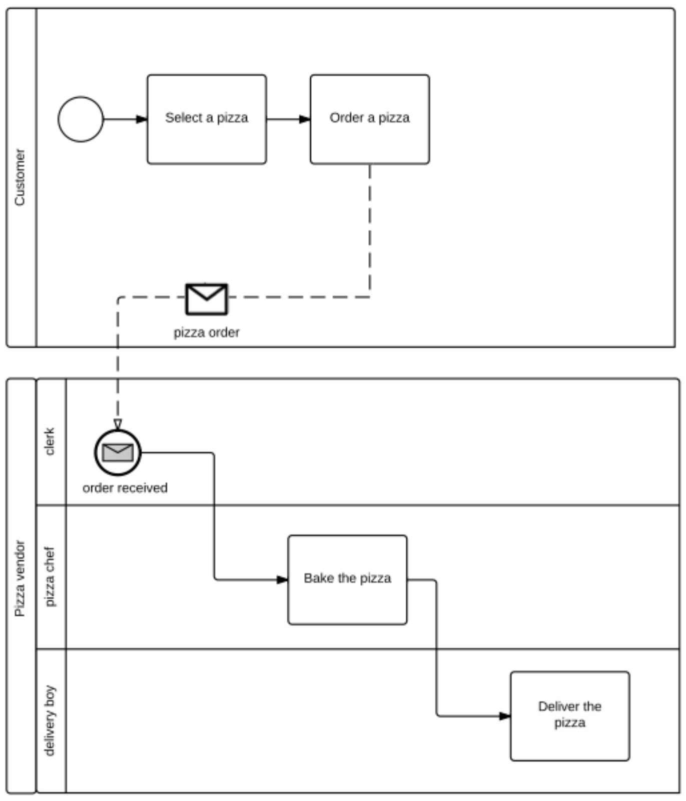


Figure 2.8: The Pizza Process: Placing an order

At this point, the Customer is back in the Pizza Process. After waiting the pizza to arrive a new communication take place between the Delivery Boy and the customer, this message, called *Pizza*, is shown in Figure 2.9. This message identifies the delivery of the Pizza to the Customer. The intermediate message event *pizza received* in the Customer lane captures the moment when the Customer actually receive the pizza from the Delivery Boy.

Yet another interaction between the Delivery Boy and the Customer should take place and is the one related to the payment of the pizza and the delivery of the recipe to the Customer. This action is shown in the tasks *Pay Pizza*, *Receive payment* and the connections between this tasks.

Finally the Customer can enjoy the pizza as shown in the task *Eat Pizza* and finally end the process by reaching his final event in his lane. The vendor's pool terminates by reaching the final event shown in the diagram, marking the end of the whole process.

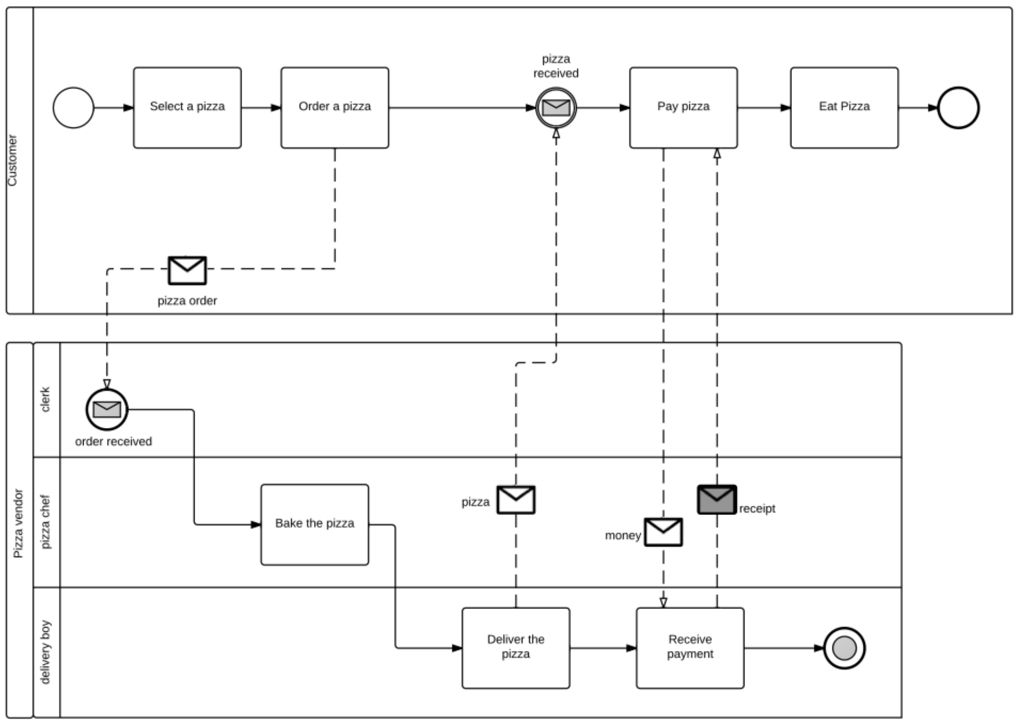


Figure 2.9: The Pizza Process: Complete Diagram

2.2.4 The BPMN XML Schema

The Open Management Group provides a definition for the Exchange Formats of the BPMN diagrams, described in [4]. The XSD, XMI and XSLT definitions are all available on-line [2] for public download and consultant.

The root element of each BPMN file must be a ¡bpmn:definitions¿ node that contains the definition of the BPMN diagram in it. BPMN files may import non BPMN files, such as other XSDs or WSDLs files if the nodes contains references to external definitions.

All BPMN elements within a file must have an *ID* attribute that is used to pass different element references. The BPMN XSD permits references by *ID* across different files using QNames, that is an optional namespace prefix and a local part. When used to make reference, the local part of the QName is expected to be the *ID* attribute of the BPMN element.

[2]http://www.omg.org/spec/BPMN/2.0/

Chapter 3

Requirements and Implementation

In this section we will describe the main components and modules in the JBPS. The most significant design and implementation issues will be briefly described in each case.

3.1 Requirements

For the JBPS we took inspiration from the Bizagi BPM Suite [1] and thus, the main requirements are heavily influenced by that piece of software. The JBPS assume that the user already have a specification of the BPMN diagrams of the software he wants to use. The BPMN diagrams should be in a .bpmn file so the software can read it and translate it to an internal ontology representation.

The JBPS should take the business model as input for the simulation. The business model should be specified in OWL, with any OWL editor, particularly we used Protégé for the creation of the example ontologies, being one of the most popular Ontology Editors available in the market.

The main assumption behind JBPS is that there are companies that specify the business model in an OWL model since they can express powerful constrains and elaborated data models with it. The model can be used to make reasoning and to check for inconsistencies that may appear while trying to model the world.

We also assume that companies have their internal and external processes modeled in BPMN, since it is one of the most important standards for modeling process in the industry nowadays. This is not a very strong assumption, since more and more companies are starting doing process modeling as they saw advantages in doing it.

Once we have the definition of the business process and the business model we have to link both somehow. Intuitively we would like to say, when task X is reached, the system should create an object O and ask the user to provide information about the properties P

and Q of that object. For this we propose a very and intuitive JSON file that will specify which are the actions that should take place in every task and which are the properties that we should provide for the object being manipulated. We will see later which is the format of this specification.

With all the configuration done the program should be able to generate a simulation of the BPMN diagram that apply the actions that we specify on each task. The user should start the JBPS and it should present all the possible process he can ran. Once a process is selected the software should guide the user through the task until he reach the completion of the process. The software should ask all necessary input to the user to populate the ontology with new individuals and new property values.

The JBPS should check all constrains every step it takes. This can be done by momentary applying the actions introduced by the user and checking if the model is still consistent. If at some point the model turns out to be inconsistent, the software should rollback the actions, presenting the user some message and asking again for the input.

All process should work with the same Ontology, this means that if we run a process that creates individuals of some kind of class, this individuals may be visible in other processes. This will give the JBPS the feel of running a working application that new data can be added, modified and deleted with the action taking place in the whole application.

With all this general requirements the JBPS was designed and build. Java was used as programming language, with some Third-Party Software Components used that we will discussed in the following section.

3.2 Design and Implementation

In this section we will describe the general design of the JBPS and the most important parts of it. Figure 3.1 shows the main components present in the global architecture. The bottom part shows the low level parts of the JBPS, dealing with the ontologies manipulation and exploration. The Engine is the central part of the architecture, it is in charge of calling different components in order to provide a consistent view of the application to the Controllers. The user interface is provided by a set of JSP that render the correct information gather from the Controllers through the Engine.

Figure 3.2 shows the most important interfaces and classes that we will describe in detail in the following subsections. This main components rely on some Jena classes to correctly implement their functionalities. When possible we try to abstract and hide the use of ontologies, this way another implementation, for example relaying on a Database

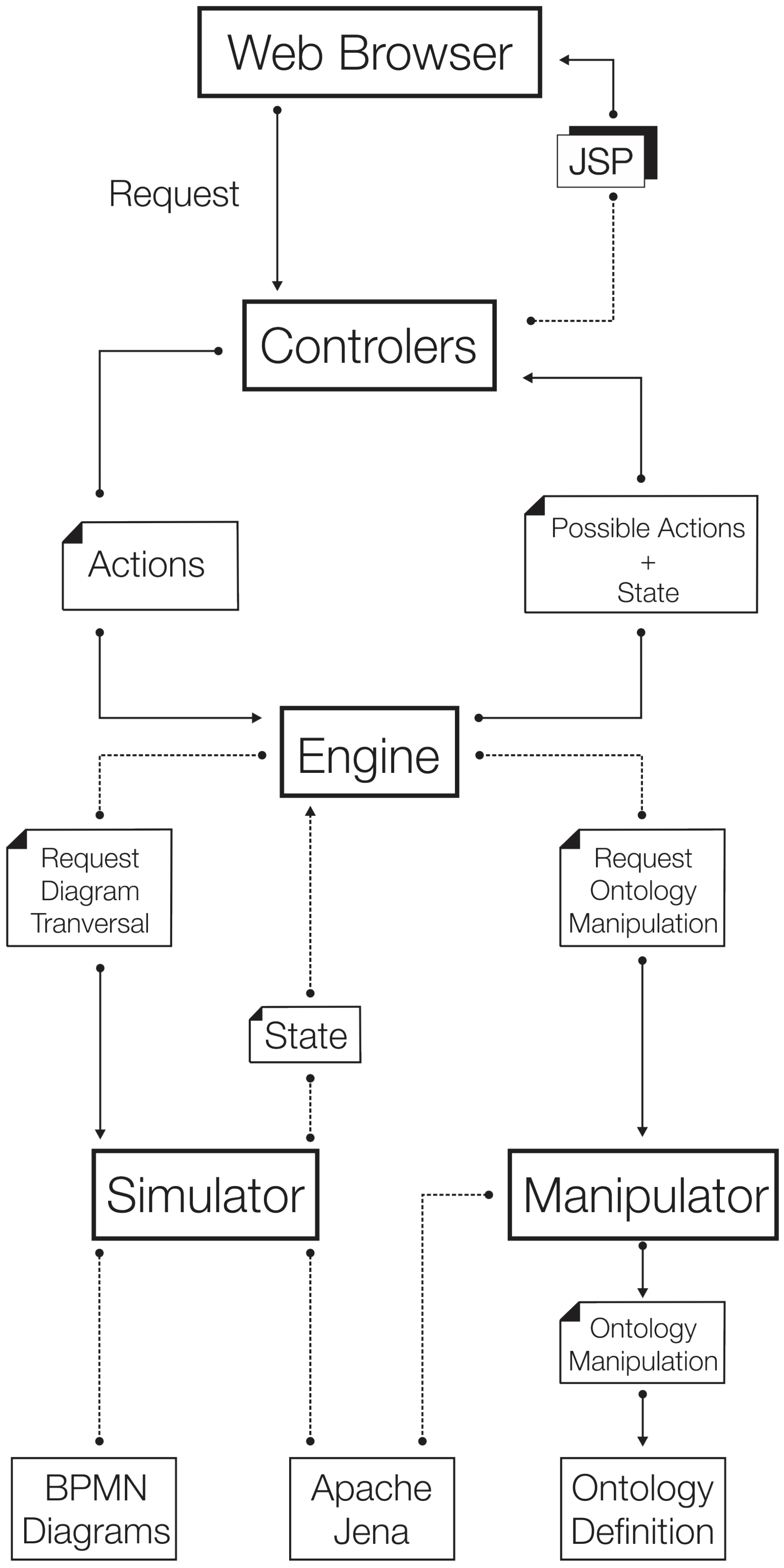


Figure 3.1: Architecture

Management System, can be provided.

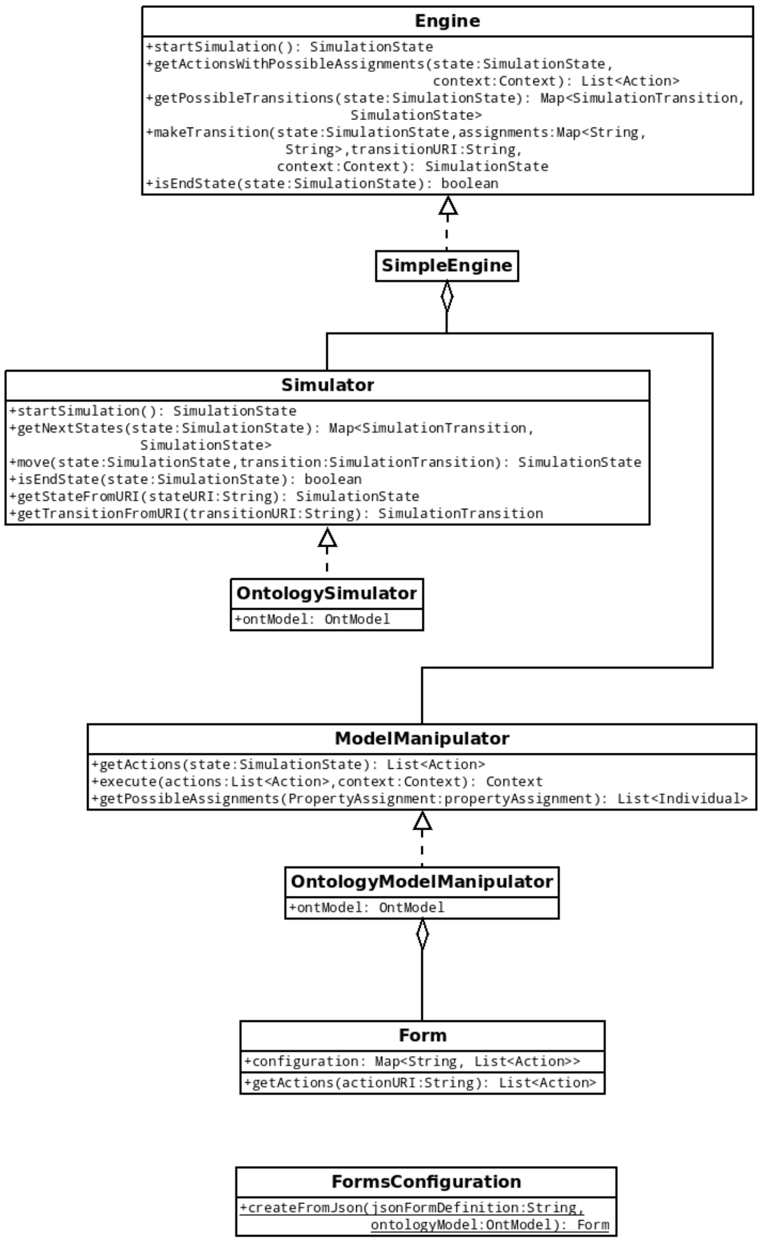

Engine
+startSimulation(): SimulationState
+getActionsWithPossibleAssignments(state:SimulationState, context:Context): List<Action>
+getPossibleTransitions(state:SimulationState): Map<SimulationTransition, SimulationState>
+makeTransition(state:SimulationState,assignments:Map<String, String>,transitionURI:String, context:Context): SimulationState
+isEndState(state:SimulationState): boolean
SimpleEngine
Simulator
+startSimulation(): SimulationState
+getNextStates(state:SimulationState): Map<SimulationTransition, SimulationState>
+move(state:SimulationState,transition:SimulationTransition): SimulationState
+isEndState(state:SimulationState): boolean
+getStateFromURI(stateURI:String): SimulationState
+getTransitionFromURI(transitionURI:String): SimulationTransition
OntologySimulator
+ontModel: OntModel
ModelManipulator
+getActions(state:SimulationState): List<Action>
+execute(actions:List<Action>,context:Context): Context
+getPossibleAssignments(PropertyAssignment:propertyAssignment): List<Individual>
OntologyModelManipulator
+ontModel: OntModel
Form
+configuration: Map<String, List<Action>>
+getActions(actionURI:String): List<Action>
FormsConfiguration
+createFromJson(jsonFormDefinition:String, ontologyModel:OntModel): Form

Figure 3.2: Main Classes

Figure 3.3 shows another important component that is the controller of the Model-View-Controller architecture. Each function in the *EngineController* corresponds to one entry point in the application, and in particular with each page that the user receive.

- *home()*: Return the name of the view that renders the home page, showing the possible lanes that the user can select for simulation

- *startSimulation()*: Given a lane, passed as argument, it returns the name of the view to render the starting point of the lane.
- *simulationState()*: This functions returns the view that should render the current state of the simulation. The view will also show possible transitions that can be taken from the state and possible actions can be taken in this particular state.
- *makeTransition()*: Execute a transition applying all actions that the user select, this will trigger the transition and return the view that will render the state that the simulation reach.

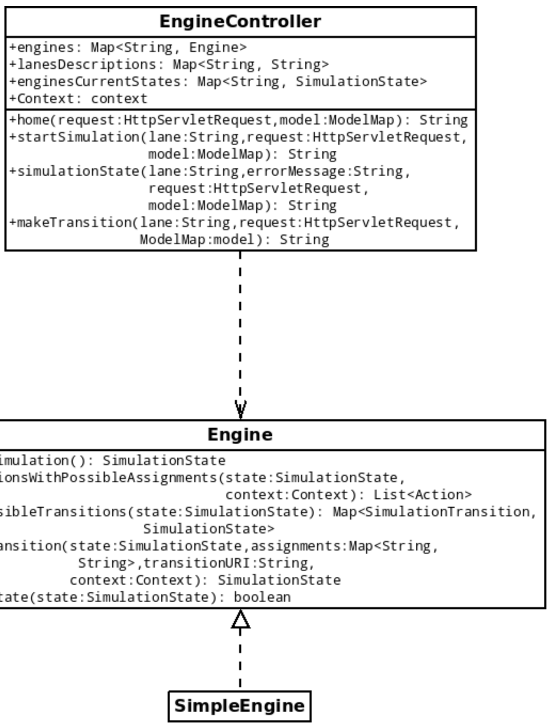

EngineController
+engines: Map<String, Engine>
+lanesDescriptions: Map<String, String>
+enginesCurrentStates: Map<String, SimulationState>
+Context: context
+home(request:HttpServletRequest,model:ModelMap): String
+startSimulation(lane:String,request:HttpServletRequest, model:ModelMap): String
+simulationState(lane:String,errorMessage:String, request:HttpServletRequest, model:ModelMap): String
+makeTransition(lane:String,request:HttpServletRequest, ModelMap:model): String
Engine
+startSimulation(): SimulationState
+getActionsWithPossibleAssignments(state:SimulationState, context:Context): List<Action>
+getPossibleTransitions(state:SimulationState): Map<SimulationTransition, SimulationState>
+makeTransition(state:SimulationState,assignments:Map<String, String>,transitionURI:String, context:Context): SimulationState
+isEndState(state:SimulationState): boolean
SimpleEngine

Figure 3.3: The Engine Controller

Similar to the *EngineController*, there is another class called *ModelController* shown in Figure 3.4. This controllers allow the user to retrieve information about the situation of the business ontology. Suppose we are creating individuals in the ontology and we want to know what we have already created, we can extract the individuals present in the ontology via this controller.

The last controller, shown in Figure 3.5, called *OntologiesController* is in charge of displaying the XML representation of the ontologies as they are in any moment in the business process execution. This maybe not very useful for the final user but we found it extremely valuable for the inspection of the derived statements the reasoner is doing

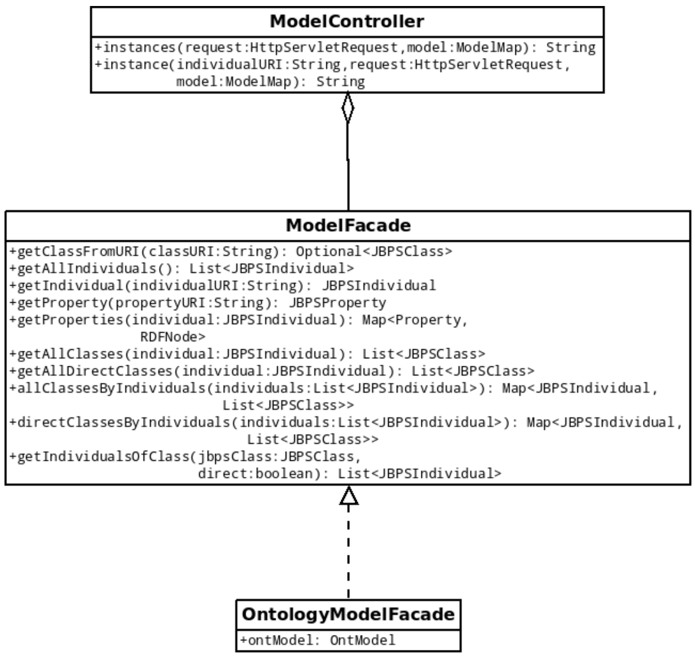

ModelController
+instances(request:HttpServletRequest,model:ModelMap): String
+instance(individualURI:String,request:HttpServletRequest, model:ModelMap): String
ModelFacade
+getClassFromURI(classURI:String): Optional<JBPSClass>
+getAllIndividuals(): List<JBPSIndividual>
+getIndividual(individualURI:String): JBPSIndividual
+getProperty(propertyURI:String): JBPSProperty
+getProperties(individual:JBPSIndividual): Map<Property, RDFNode>
+getAllClasses(individual:JBPSIndividual): List<JBPSClass>
+getAllDirectClasses(individual:JBPSIndividual): List<JBPSClass>
+allClassesByIndividuals(individuals:List<JBPSIndividual>): Map<JBPSIndividual, List<JBPSClass>>
+directClassesByIndividuals(individuals:List<JBPSIndividual>): Map<JBPSIndividual, List<JBPSClass>>
+getIndividualsOfClass(jbpsClass:JBPSClass, direct:boolean): List<JBPSIndividual>
OntologyModelFacade
+ontModel: OntModel

Figure 3.4: The Model Controller

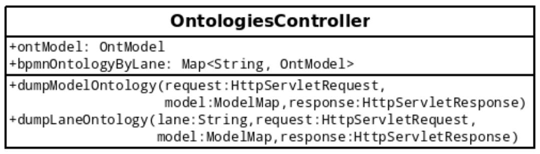

OntologiesController
+ontModel: OntModel
+bpmnOntologyByLane: Map<String, OntModel>
+dumpModelOntology(request:HttpServletRequest, model:ModelMap,response:HttpServletResponse)
+dumpLaneOntology(lane:String,request:HttpServletRequest, model:ModelMap,response:HttpServletResponse)

Figure 3.5: The Ontologies Controller

while we navigate the BPMN diagrams executing the actions.

Finally, Figure 3.6 the most important entities defined in the JBPS. Some of the classes, like *JBPSProperty*, *JBPSClass* and *JBPSIndividual* are Wrappers of other ontology classes, providing additional functionalities. Other classes like *Context* are structural and serve to hide concepts as the variable assignment in the process transitions.

3.2.1 The Simulator Interface

The classes that implements the *Simulator* interface are responsible for the navigation of the BPMN model. The interface is thought to be stateless, in the sense that states in the BPMN model is always passed and returned to all functions, this gives the advantage of rendering the interface functional and thus allow us to reason very simply in terms of state transition. The definition of the Simulator interface is the following:

```
public interface Simulator {
}
```

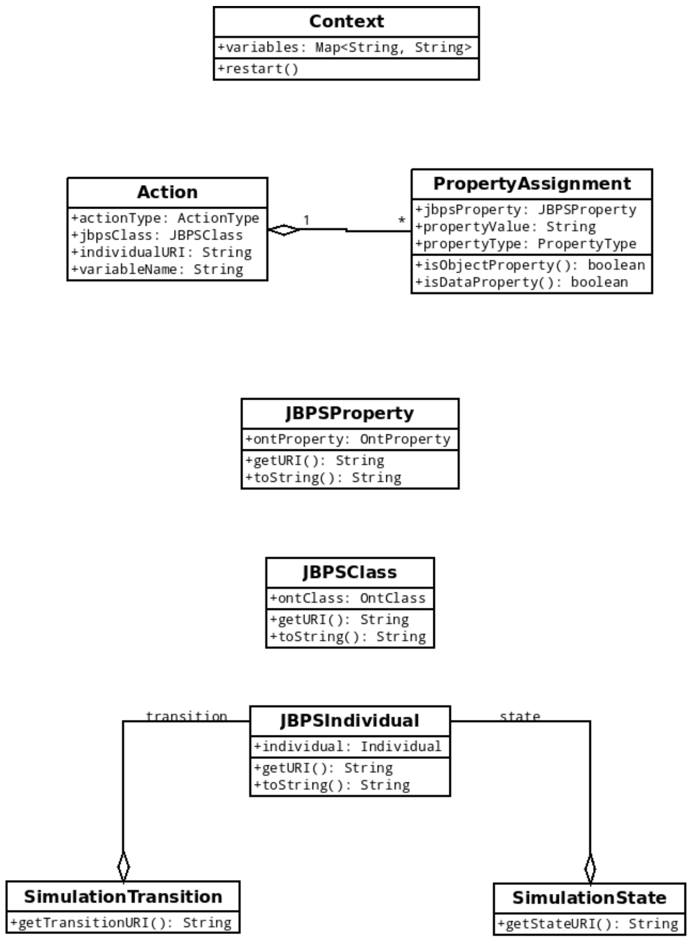


Figure 3.6: JBPS Entities

```
    SimulationState startSimulation();

    Map<SimulationTransition, SimulationState> getNextStates(
        SimulationState state);

    SimulationState move(SimulationState state,
                         SimulationTransition transition)
        throws BPMNInvalidTransition;

    boolean isEndState(SimulationState state);

    SimulationState getStateFromURI(String stateURI);

    SimulationTransition getTransitionFromURI(String transitionURI);
}
```

The function *startSimulation()* provides the entry point of the BPMN lane, returning a *SimulationState* object that represents the starting point within the lane.

getNextStates() returns a mapping that represent all the transitions that can be done in the current state, passed as an argument to the function, to all the states that are reached when executing this transitions. This is used for the user to see the actual transition and the final state before it can take a decision which path to follow in the BPMN model traversal.

The function *move()* takes a state in the BPMN model and a possible transition and execute that transition from the state, returning the final reached state. This function allows us to navigate the BPMN model as the user selects the different transitions while using the prototype. This functions may throw a *BPMNInvalidTransition* if the transition cannot be take from the provided state. This may be because there is no such transition from the provided state or because the state, or the transition, does not exist in the BPMN model.

isEndState() as it names said, allows to check if some state is a final state or not. The other functions *getStateFromURI()* and *getTransitionFromURI()* provide, respectively, access to the objects *SimulationState* and *SimulationTransitions* given the corresponding URI.

Only one implementation of the *Simulator* interface is provided, the *OntologySimulator* class, that makes heavy use of the Jena *OntModel* class in order to implement the necessary functionalities.

```
public class OntologySimulator implements Simulator {

    private final OntModel bpmnOntologyModel;

    public OntologySimulator(OntModel bpmnOntologyModel) {
        this.bpmnOntologyModel = bpmnOntologyModel;
    }

    ...
}
```

3.2.2 The ModelManipulator Interface

The *ModelManipulator* interface is used for manipulating the Model representing the business objects. In our case the Model will be the ontology used to describe the business objects with it's classes, individuals, properties and constrains. The definition of the *ModelManipulator* interface is the following:

```
public interface ModelManipulator {

    List<Action> getActions(SimulationState state);

    Context execute(List<Action> actions, Context context)
        throws InvalidPropertyAssignment;

    List<Individual> getPossibleAssignments(
        PropertyAssignment propertyAssignment);
}
```

The function *getActions()* takes a *SimulationState* object that represents a state in the BPMN model and returns a list of *actions* that can be taken in that state.

An *Action* is something that involves an individual of a class in the underlying business model. It can be a creation of an individual of a particular class in the business model, for example, in the *Pizza Business Process* described in section 2.2.3 an action can be the creation of the Order that the customer make when calling the Pizza Vendor, if the ordering pizza process was supported b y a software. An action can be a deletion of a particular individual in the model or a modification of the individual as well, giving new values for the properties that it has.

execute() takes a list of actions and a context and apply the actions to the underlying ontology model. The list of actions specify what to do with the individuals of the model. The context contains the information related to the variables that can be defined in the Ontology. For example, suppose we create an individual in some state of the BPMN model and we want to update some values of the properties of that individual in a different state. We define a variable that points to the created individual and then we can make reference to that individual in other stages of the BPMN transition in order to delete or alter that individual. The information about the variable and the individual pointed by that variable is stored in an *Context* object that is passed and returned by the *execute()* function.

This function may throw a *InvalidPropertyAssignment* exception if the actions passed as parameters violate some constrains in the underlying ontology model. The Pellet reasoner, described in section 4.2, was particularly useful in the implementation of this functionality. The Jena reasoners does nos support some kind of constrains checks, like disjoin properties, so it was impossible to correctly implement the throwing of the *InvalidPropertyAssignment* exception in some cases. Fortunately the Pellet reasoner was easily incorporated in Jena.

For the implementation of the *execute()* function, an attempt to execute all actions is done and a control of the consistency of the model is performed. If the model became inconsistent, a rollback is performed and the *InvalidPropertyAssignment* exception is thrown with an inner message generated by the Pellet reasoner that explains the origin of such inconsistency. If the model is still consistent after the actions take place, the model changes incorporating the new individuals or property values.

Finally, the function *getPossibleAssignments()* takes an instance of *PropertyAssignment* and returns a list of all possible individuals that can be assigned to the property associated to the *PropertyAssignment* object. *PropertyAssignment* represents a possible assignment that can be done to a property in the underlying business ontology model. This function basically return the individuals that are of a class that are part of the range of the underlying property. This are the individuals that the user of the JBPS can choose when assigning values to the properties.

The implementation of the *ModelManipulator* interface is the *OntologyModelManipulator* class. This class as the *OntologySimulator* class use the Jena *OntModel* to perform all the operations. Since some of the functions in the interface require the creation of new individuals in the model, the names of the individuals are generated automatically but a base URI for the naming is passed as an argument to the constructor of the class. It does also receive a *Form* object that represent which are the actions that can be taken in each state, this object is described in the following section.

```
public class OntologyModelManipulator implements ModelManipulator {

    private final OntModel ontologyModel;
    private final Form form;
    private final String baseURIForNameing;

    public OntologyModelManipulator(OntModel ontologyModel,
                                    Form form,
                                    String baseURIForNameing) {
        this.ontologyModel = ontologyModel;
        this.form = form;
        this.baseURIForNameing = baseURIForNameing;
    }

    ...
}
```

3.2.3 The Form and the FormsConfiguration class

The *Form* and *FormsConfiguration* classes provides the functionality for reading the configuration files that describe which are the actions that can be performed on the business ontology model on each particular BPMN diagram node.

The configuration of the different actions is specified JSON format and describe in a very intuitive and human readable way what actions are taken in each node. Continuing with the *Pizza Business Process* example described in section 2.2.3 a possible configuration may be the following:

```
{
  "forms": [
    {
      "form": {
        "state": "...#orderPizza",
        "actions": {
          "insert": [
            {
              "classURI": "...#order",
              "propertyValues": [
                {
                  "propertyType": "objectProperty",
                  "propertyURI": "...#orderAddress"
                },
                {
                  "propertyType": "objectProperty",
                  "propertyURI": "...#orderedPizza"
                }
              ]
            }
          ]
        }
      }
    }
  ]
}
```

This configuration specifies that in the *orderPizza* task a new individual of the class *order* should be create, specifying the properties *orderAddress* and *orderedPizza* that are both object properties. This configuration is read by this 2 classes and is used during the BPMN diagram traversal to provide the actions in each step as discussed in the previous sections.

3.2.4 The Engine Interface

The *Engine* interface provides an abstraction for the whole process of traversing the BPMN model and modify the underlying business model. The definition of the *Engine* interface is the following:

```
public interface Engine {

    SimulationState startSimulation();

    List<Action> getActionsWithPossibleAssignments(
        SimulationState state, Context context);

    Map<SimulationTransition, SimulationState> getPossibleTransitions(
        SimulationState state);

    SimulationState makeTransition(SimulationState state,
                                   Map<String, String> assignments,
                                   String transitionURI,
                                   Context context)
        throws InvalidPropertyAssignment, BPMNInvalidTransition;

    boolean isEndState(SimulationState state);
}
```

Functions *startSimulation()*, *isEndState()* and *getPossibleTransitions()* have the same semantic as the corresponding *Simulator* interface's functions. The function *makeTransition()* encapsulates the behavior of the *move()* function of *Simulator* and the function *execute()* of *ModelManipulator* in a single method. Finally, the *getActionsWithPossibleAssignments()* function returns the possible actions that can be performed in a particular state given a context.

The *SimpleEngine* class implements the *Engine* simulator functions delegating most of the functionalities to the underlying *Simulator* and *ModelManipulator* classes.

```
public class SimpleEngine implements Engine {

    private final Simulator simulator;
    private final ModelManipulator manipulator;
    private final ModelFacade modelFacade;

    public SimpleEngine(Simulator simulator,
                        ModelManipulator manipulator,
                        ModelFacade modelFacade) {
        this.simulator = simulator;
        this.manipulator = manipulator;
        this.modelFacade = modelFacade;
    }

    ...
}
```

3.2.5 Testing

A big amount of test was used to validate the different functionalities of all the interfaces. Tests where done using the interfaces instead of the implementations, this way different implementations can use the same tests to validate their correctness.

This was accomplished using JUnit v4 and defining the test cases as abstract, with an abstract function that provides the corresponding instance of the interfaces. For example, for the *Simulator* interface, an abstract test case called *SimulatorTest* was defined with an abstract function that provides the correct implementation of the *Simulator* interface. All test are then defined using the abstract function to get *Simulator* implementation, as can be show in the following piece of code:

```
public abstract class SimulatorTest {

    protected abstract Simulator getSimulator(String resource)
        throws IOException;

    @Test
    public void getStateFromURIWhenExists() throws IOException {

        Simulator simulator =
            getSimulator(getResource("SimplePurchaseRequestBPMN"));

        ...
    }

    ...
}
```

After that a concrete implementation of the abstract class is created providing the correct interface implementation. Continuing with the example, in the *SimulatorTest* test case, a concrete test case called *OntologySimulatorTest* is created that provides the *OntologySimulator* implementation of the *Simulator* interface. This way all test defined in the *SimulatorTest* test case are lunched with that concrete interface implementation as can be shown in the following definition:

```
public class OntologySimulatorTest extends SimulatorTest {
    ...

    @Override
    protected Simulator getSimulator(String resource) {
        OntModel bpmnOntology = getOntologyFromFile(resource);
        return new OntologySimulator(bpmnOntology);
    }

    ..
}
```

If a new implementation of the interface is developed, the abstract test can be used to test the correctness of the implementation. This way we guarantee that all the implementations of an interface provides a common logic.

Chapter 4

Third-party Software Component

In this section we will describe some of the framework, libraries and tools that we have chosen in the development of the Java Business Process Simulator. Among the most important choice we will describe the *Jena Framework*, the *Pellet Reasoner*, the *Camunda Eclipse Plugin* and the *Spring Framework*.

It is also worth mentioning that since the development of the prototype was done using a Test-driven development (TDD) approach the JUnit v4 framework was extensively used.

4.1 Apache Jena

Apache Jena[1] is an Open Source Java Framework aimed to build Semantic Web applications. It provides a solid API for creating and manipulating RDF graphs, RDFS and OWL ontologies. Originally developed in the HP Labs at Bristol in 2000 the Jena project was adopted by the Apache Software Foundation at the end of 2010. A big amount of documentation about the Apache Jena Framework can be found in on-line in the Apache Jena web site[2].

Apache Jena abstract an RDF graph using an abstraction called *Model*. The Model interface provides functions for manipulating Resources, Properties and Literals. It also expose powerful searching functions in the Model using the *Statement* and *Selector* interfaces.

Jena provides operations for manipulating Models as a whole, these are the common set operations of union, intersection and difference. It is also possible to add all the statements of one Model to another, a feature that we will use in our prototype to try-and-rollback for checking Model consistency when the model is manipulated.

[1]http://jena.apache.org/

[2]http://jena.apache.org/documentation/

RDF defines a special kind of resources aimed to represent collections of either literals or resources. There are three kind of container, all of them supported by Jena:

- *Bag*: An unordered collection.
- *Alt*: An unordered collection intended to represent alternatives.
- *Seq*: An ordered collection.

Jena does provide support for OWL mainly by extending the Model interface creating the *OntModel* interface. Since the inferred tuple are generating while manipulating the OntModel, more or less triples would be shown in it depending on the OntModel capability of the underlying implementation. The *InfModel* interface, an intermediate interface between Model and OntModel, provides a functionality for checking the validity or inconsistency of the Ontology. The result of this functionality will depends on the implementation of the Reasoner used.

Among the various features offered by Jena, it provides several RDF serialization of the Models:

- Relational Database
- RDF/XML
- Turtle
- Notation 3

Apache Jena does provide support for SPARQL (SPARQL Protocol and RDF Query Language), a query language for executing queries on RDF graphs. SPARQL is a standard for querying RDF graphs developed by the RDF Data Access Working Group (DAWG) and the World Wide Web Consortium (W3C).

Among the various options apart from Apache Jena we can find *Sesame*[3], an Open Source framework for querying and analyzing RDF data. Sesame does not natively provide support for OWL although there are some extensions that offer some Jena similar capabilities. Jena was chosen over Sesame since Jena is much more supported and user in the industry and nowadays is like in standard when developing OWL based applications.

JRDF[4] is yet another Java Open Source RDF Framework that provides an Object Oriented model for the manipulation of RDF graphs. Since JRDF does not currently provide support for OWL it was not considered for usage in our prototype.

[3]http://www.openrdf.org/
[4]http://jrdf.sourceforge.net/

4.2 Pellet

Pellet[5] is an complete OWL reasoner that provides a standard and powerful reasoner with support individual and user-defined data-types. A description of the Pellet Reasoner and it's implementation can be found in here [6]

While developing the JBPS we try to force some *disjoint properties* checks while reasoning about Ontologies. Since disjoint properties are part of OWL2, the checks continue to fail while proving with different reasoning services provided by Apache Jena that does not cover OWL2. Because of this we discard all the build in Jena reasoners and we adopt the Pellet Reasoner. Thanks to the Apache Jena flexibility it was pretty straightforward to use the Pellet Reasoner within the Jena Framework.

4.3 Spring

The Spring Framework[6] is an open source framework for the development of Java applications and an inversion of control container. The framework initially lunched in 2002 is a becoming a standard in the Java world for application development.

The Spring Framework contains a variety of modules, each one specialized in different proposes, that implements different aspect of modern development standards. The different Spring modules are the followings:

- Inversion of control container: Used for the configuration of the various components in the application and the life-cycle management of different objects. This is mainly achieved via dependency injection done by the Spring Framework.
- Model View Controller: This component provide a way to extend and customize web applications and RESTfull web services using the *Model View Controller* software architecture pattern. This pattern is used in the separation of the representation of the information and the user interaction with it.
- Data access to different Relational Database Management Systems.
- Transactional management:
- Aspect oriented programming that allow the implementation of cross-cutting concerns in aspects.
- Remote access framework with RPC style.
- Unit and Integration Testing.

[5]http://clarkparsia.com/pellet
[6]http://www.springsource.org/

- Authorization and Authentication.
- Messaging with message queues via JMS.
- Convention over configuration.

The *Inversion of control container* module is heavily in the configuration and dependency injection of the different components of the JBPS. The *Inversion of control container* module provides a consistent way of configuring applications making use of reflection. Dependency injection is a software pattern that allows the container to provide objects to other objects, via either constructors, properties, or factory methods. This way objects are developed without taking care of the instantiation of other objects that may be dependencies and leave this responsibility for the container that takes care of that.

The other component used in the JBPS is the *Model View Controller* module. Different objects are created for the implementation of a transparent Model-View-Controller architecture.

4.4 Maven

Maven[7] is a tool for the management and construction of Java Projects. Similar to *Apache Ant* it provides a simpler configuration construct based on XML files. Maven used Project Object Model (POM) for describing the project to be constructed, it's dependencies and external resources needed for the construction of the project.

A marvelous characteristic of maven is that is prepared to work in-line, downloading different plugins and dependencies from a central Maven repository. Dependencies are downloaded in the local repository, for further use, and are resolved recursively, creating the complete structure of the needed libraries or projects. Plugins are used to execute different operations within Maven, for example, there are plugins to build Eclipse projects out of the POM description for the projects and other plugins to lunch the Jetty HTTP server.

4.5 Others

There are some others libraries chosen in order to render the code shorter and more standard. Among the various libraries used in the JBPS there is a intensive use of the Google Guava Library[8], more precisely for the easy instantiation of the principal collection and mapping interfaces.

[7]http://maven.apache.org/
[8]http://code.google.com/p/guava-libraries/

The Jackson Library[9] is used for the parsing of the JSON configuration files that are used to specify the input specification of the JBPS prototype.

[9]https://github.com/FasterXML/jackson

Chapter 5

Study Cases

In this section we will show different study cases that we used while developing the JBPS prototype in order to confront our ideas with real world problems and applications.

5.1 The Purchase Order Process

We will start describing a very simple Purchase Order business process to illustrate some of the components and the configuration of the JBPS. Let's start describing the purchase order process described in figure 5.1.

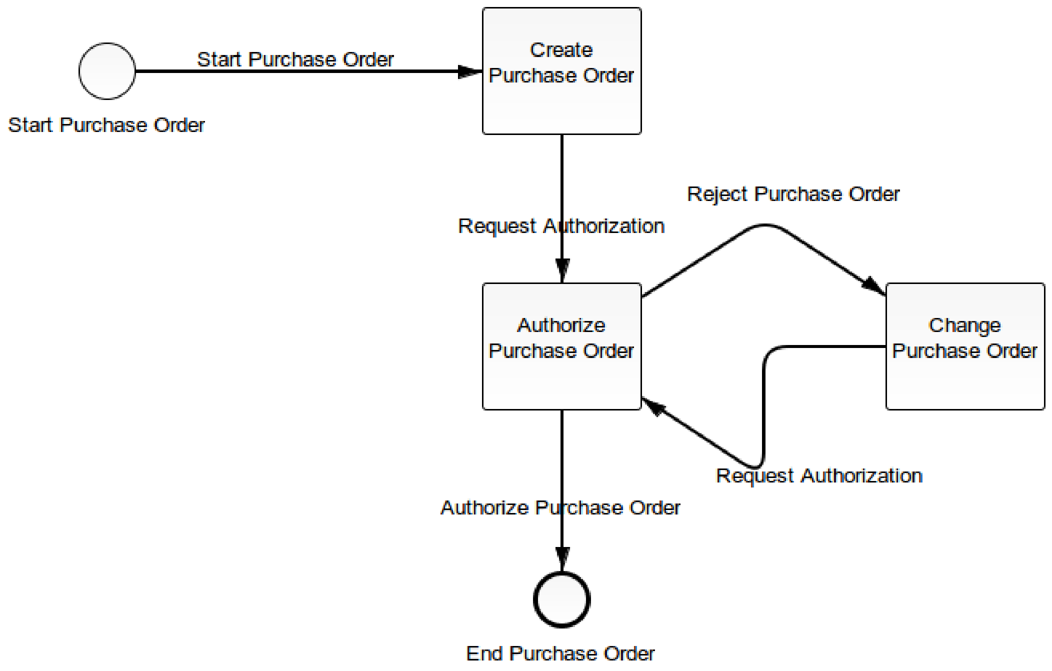


Figure 5.1: Purchase Order Diagram

The diagram shows a starting event called *Start Purchase Order* that is the entry point of the BPMN process. This event is connected to the *Create Purchase Order* that represent the creation of the Purchase Order in the process.

After that the process continues through the *Request Authorization* sequence flow to the *Authorize Purchase Order* task. Here two things can happens, or the purchase order is authorized via the *Authorize Purchase Order* sequence flow, reaching the end event *End Purchase Order*, or the purchase order is not authorized and then sequence flow *Reject Purchase Order* is taken, ending in the *Change Purchase Order* task. In this task the purchase order should be changed and submitted again for approval via the *Request Authorization* sequence flow.

This is a very simple process that any company can have regarding the submission a purchase order. This diagram does not talk about which are the information needed in each task and which are the objects that are created, manipulated or deleted thought the BPMN diagram.

We also need to define the underlying ontology that will represent the business model we will be using with this BPMN diagram. The statements will be described in OWL since it is easier to understand given a basic logical background. We will start by defining the classes that we will use, in our case we will have people that are somehow related to the company:

$$Person \tag{5.1}$$

$$PurchaseRequest \tag{5.2}$$

We will also have two properties, (5.3) makes reference to the fact that a Purchase Request have a person associated and (5.4) captures the idea that a Purchase Request have a person that is responsible for the flow of the purchase order.

$$purchaseRequestClient \tag{5.3}$$

$$purchaseRequestResponsible \tag{5.4}$$

For the properties we will specify that both have the class *PurchaseRequest* as domain and the class *Person* as range. This is done with the following two definitions:

$$purchaseRequestClient : PurchaseRequest \rightarrow Person \tag{5.5}$$

$$purchaseRequestResponsible : PurchaseRequest \rightarrow Person \tag{5.6}$$

Also we would like to state that this two properties are functional, meaning that for each element in the domain, a *PurchaseRequest*, there is at most one element in the range, a *Person*. The fact that this two properties are functional is stated in the following way:

$$\top \sqsubseteq \leq 1\, purchaseRequestClient \tag{5.7}$$

$$\top \sqsubseteq \leq 1\, purchaseRequestResponsible \tag{5.8}$$

Another statement worth noticing is the fact that the properties *purchaseRequest-Client* and *purchaseRequestResponsible* should be disjoint. This means that given a single *PurchaseRequest* the associated *Person* of each property cannot be the same, thus, the client of a purchase request and the responsible of the purchase request cannot be the same. This property seems more than reasonable in any business context and is specified in the following way:

$$DisProp(purchaseRequestClient, purchaseRequestResponsible) \tag{5.9}$$

Finally let's include some individuals in the classes we defined previously, this part should go in the A-Box section.

$$Person(employee) \tag{5.10}$$

$$Person(client) \tag{5.11}$$

$$Person(randomPerson) \tag{5.12}$$

This statements say that there are some individuals in the *Person* class and that we will use the names *employee*, *client* and *randomPerson* to denote some individuals in that class. It is important to notice that we are not saying that this names refer to different individuals, we are merely saying that there are three names that represent individuals in the *Person* class, but the individual that they represent can be the same.

Here we give the definition of the classes, relations, individuals and statements using a logical language but in the practice we will use software are Protégé[1] that provides a graphical interface to define ontologies. The advantage of this kind of software is not only the graphical representation that for big ontologies may be very important, but also the fact that ontologies can be checked for inconsistency before starting to use them in JBPS. Also ontologies are saved in .owl files, that can be used to feed the JBPS.

The next step is to define what to do in each state of the BPMN diagram. We will state that in the createPurchaseOrder task, an insertion should be done, and that insertion should correspond to an object of the class *PurchaseRequest*. For this object we should provide a value for the properties *purchaseRequestClient* and *purchaseRequestResponsible*. All this is done with the following form definition:

```
"form": {
  "state": "...#createPurchaseOrder",
  "actions": {
    "insert": [
      {
        "variableName": "purchaseOrder",
        "classURI": "...#PurchaseRequest",
        "propertyValues": [
          {
            "propertyType": "objectProperty",
            "propertyURI": "...#purchaseRequestClient"
          },
          {
            "propertyType": "objectProperty",
            "propertyURI": "...#purchaseRequestResponsible"
          }
        ]
      }
    ]
  }
}
```

We also associate a variable name ”purchaseOrder” to that instance, this will allow us to identify this previously created object to update some property values or to delete the entire object later in the diagram.

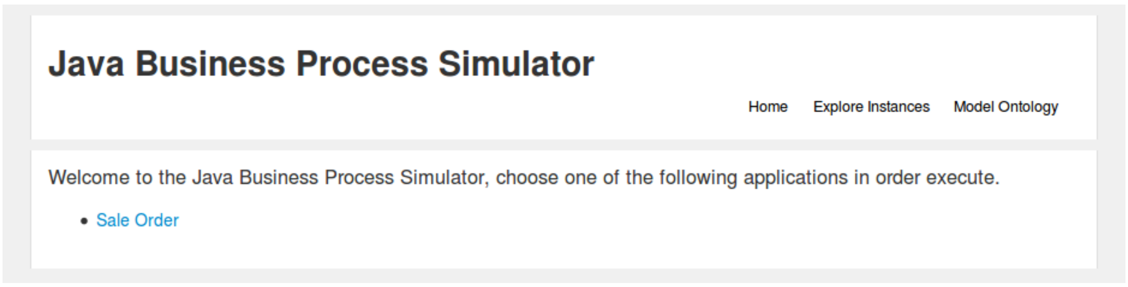
In the *changePurchaseOrder* task the user should edit the already created purchase order and all the properties should be allowed to be modified. This is specify with the following definition:

[1]http://protege.stanford.edu/

```
"form": {
  "state": "...#changePurchaseOrder",
  "actions": {
    "update": [
      {
        "variableName": "purchaseOrder",
        "classURI": "...#PurchaseRequest",
        "propertyValues": [
          {
             "propertyType": "objectProperty",
             "propertyURI": "...#purchaseRequestClient"
          },
          {
            "propertyType": "objectProperty",
            "propertyURI": "...#purchaseRequestResponsible"
          }
        ]
      }
    ]
  }
}
```

Note we use the same variable to make reference to the same instance we want to update. This will present the same ontology individual in the different state with the same properties assignments the user can modify it.

Once we have all this definitions we can give them as input to the JBPS to generate the process simulation. After initialization we can access the simulator using a web browser. The first page we see is the one shown in figure 5.2. We will have one option for each lane we had defined, in our case only the *Sale Order* lane is shown.

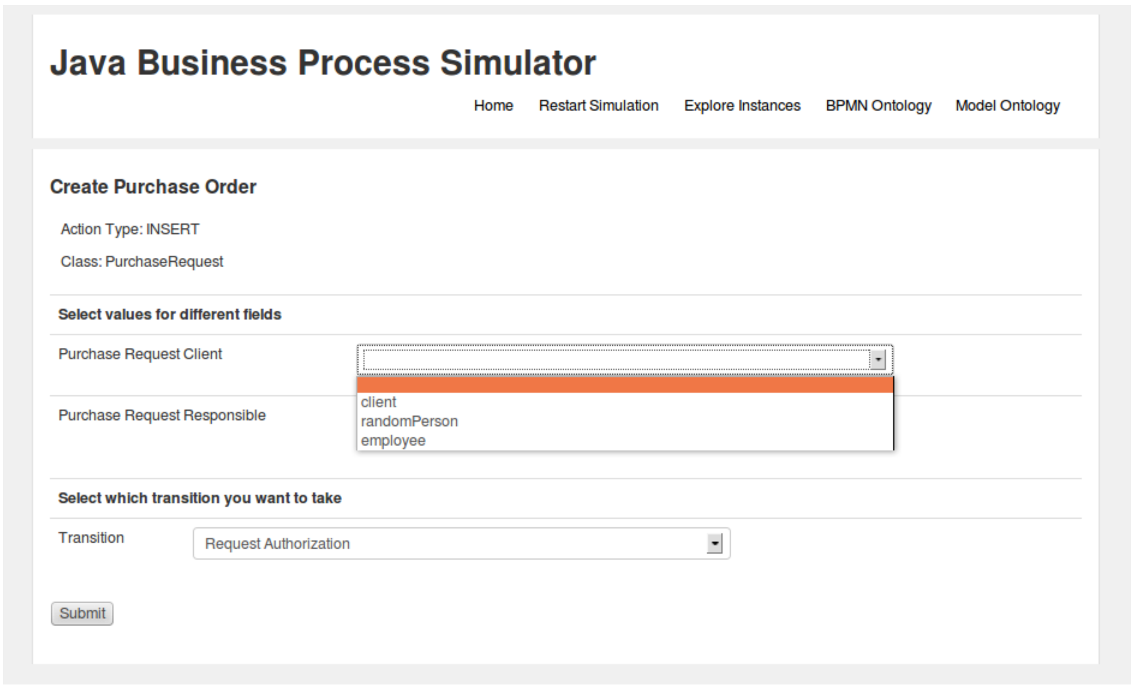


Welcome to the Java Business Process Simulator, choose one of the following applications in order execute.

- Sale Order

Figure 5.2: Purchase Order: Start simulation

After selecting the *Sale Order* lane the BPMN diagram traversal starts and reach the *Create Purchase Order* task. Associated with this task a form action, defined in JSON,


Java Business Process Simulator

Home Restart Simulation Explore Instances BPMN Ontology Model Ontology

Create Purchase Order

Action Type: INSERT

Class: PurchaseRequest

Select values for different fields

Purchase Request Client	client / randomPerson / employee
Purchase Request Responsible	

Select which transition you want to take

Transition	Request Authorization

Submit

Figure 5.3: Purchase Order: Creation 01

takes place, and we see a page similar to Figure 5.3.

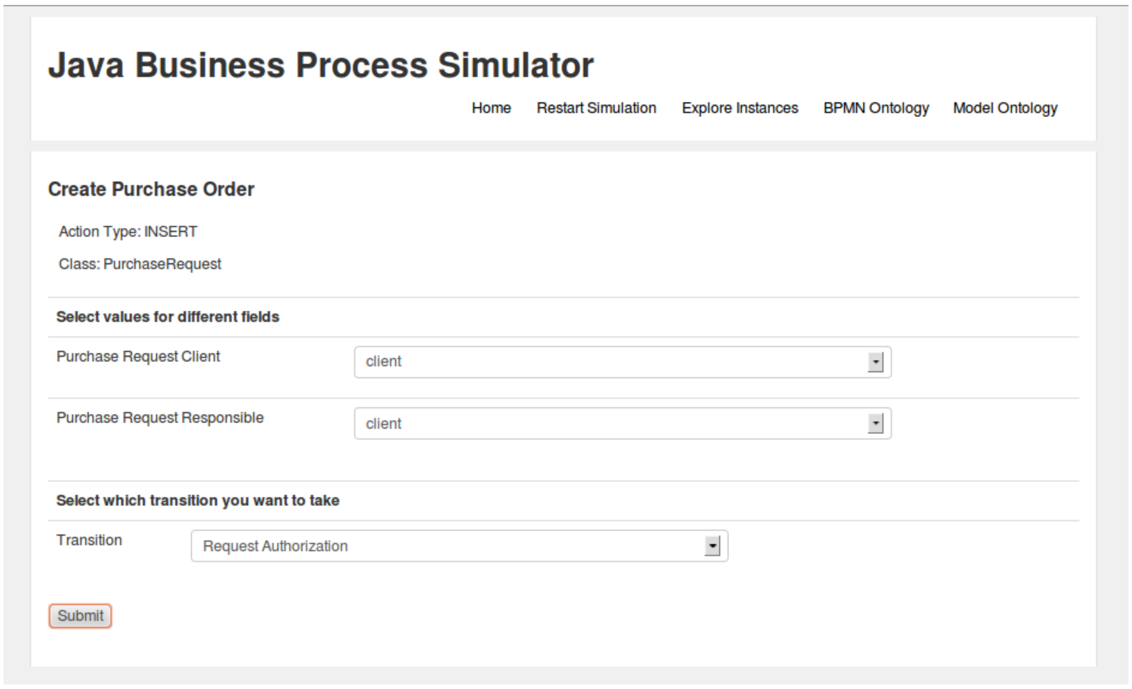
Here we can choose which is the Client and which is the Responsible for the Purchase Order we are about to create. The fields are populated with the individuals we have in the Ontology. Another field called *Transition* is shown in the figure, here we can choose which transition we would like to take, in this case only one transition is possible, this is the *Request Authorization* Transition.

Suppose we would like to choose the same individual for both the Client and Responsible field, as show in figure Figure 5.4.

After submitting the form, the JBPS would return to the same screen, showing us a message saying that the transition cannot be taken and we are presented with the same screen as before. The message shown is taken from the Pellet reasoner, this should explain why the transition was not accomplished.

In our case, the transition cannot take place since we are assigning the same individual to both the *purchaseRequestClient* and the *purchaseRequestResponsible* properties. This makes the ontology inconsistent with the axion 5.9 defined previously.

We can correct our assignment selecting the individual *employee* for the Responsible field and submit the form. Next we would see something like Figure 5.6. We are now


Java Business Process Simulator

Create Purchase Order

Action Type: INSERT

Class: PurchaseRequest

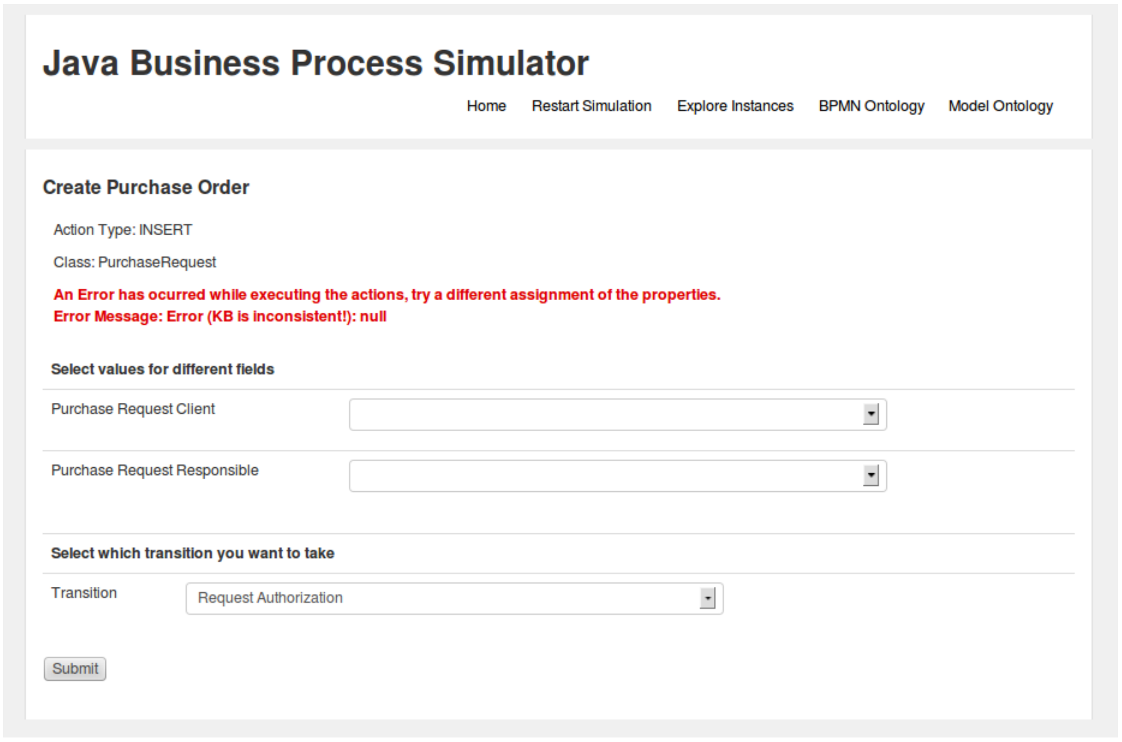
Select values for different fields

Purchase Request Client	client
Purchase Request Responsible	client

Select which transition you want to take

Transition	Request Authorization

Figure 5.4: Purchase Order: Creation 02


Java Business Process Simulator

Create Purchase Order

Action Type: INSERT

Class: PurchaseRequest

An Error has ocurred while executing the actions, try a different assignment of the properties.
Error Message: Error (KB is inconsistent!): null

Select values for different fields

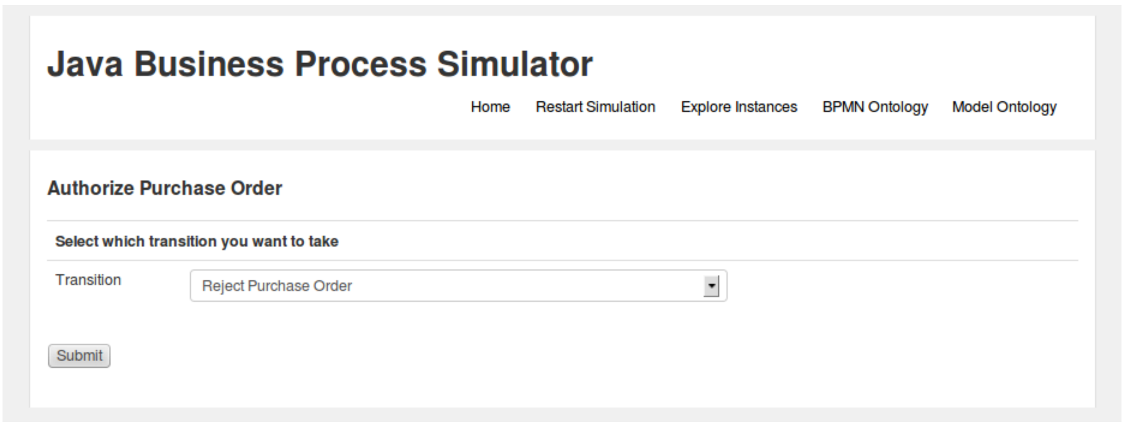
Purchase Request Client	
Purchase Request Responsible	

Select which transition you want to take

Transition	Request Authorization

Figure 5.5: Purchase Order: Constraint Violation

in the *Authorize Purchase Order* task, where no action was defined and two possible


Java Business Process Simulator

Home Restart Simulation Explore Instances BPMN Ontology Model Ontology

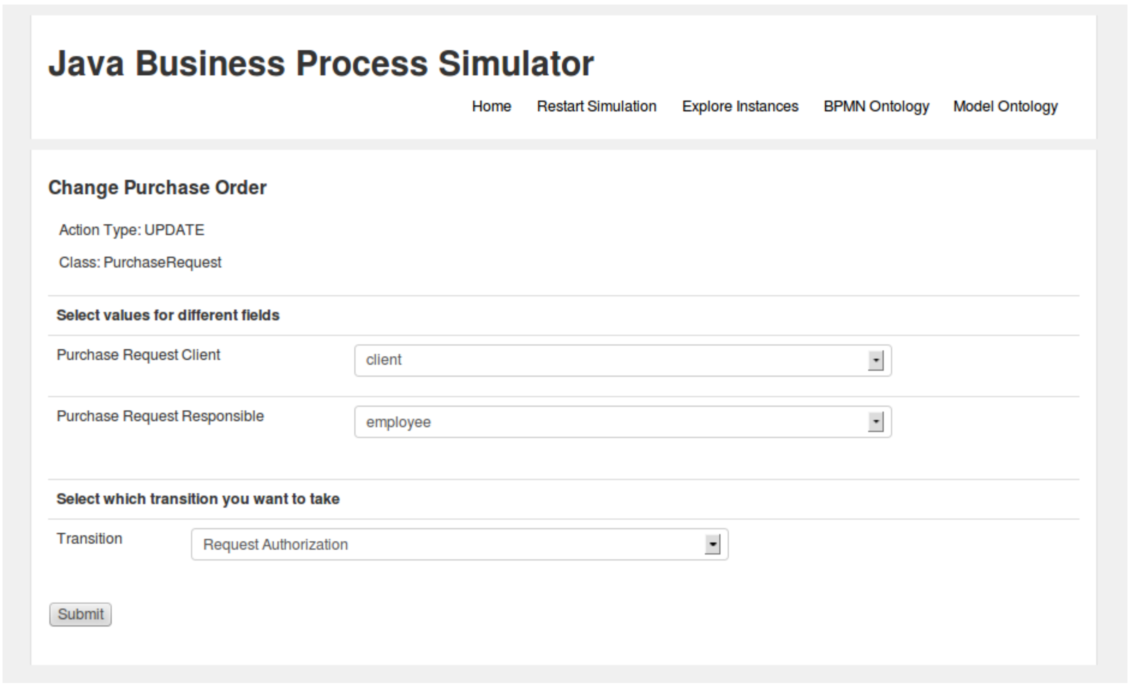
Authorize Purchase Order

Select which transition you want to take

Transition: Reject Purchase Order

Submit

Figure 5.6: Purchase Order: Reject Purchase Order


Java Business Process Simulator

Home Restart Simulation Explore Instances BPMN Ontology Model Ontology

Change Purchase Order

Action Type: UPDATE

Class: PurchaseRequest

Select values for different fields

Purchase Request Client	client
Purchase Request Responsible	employee

Select which transition you want to take

Transition: Request Authorization

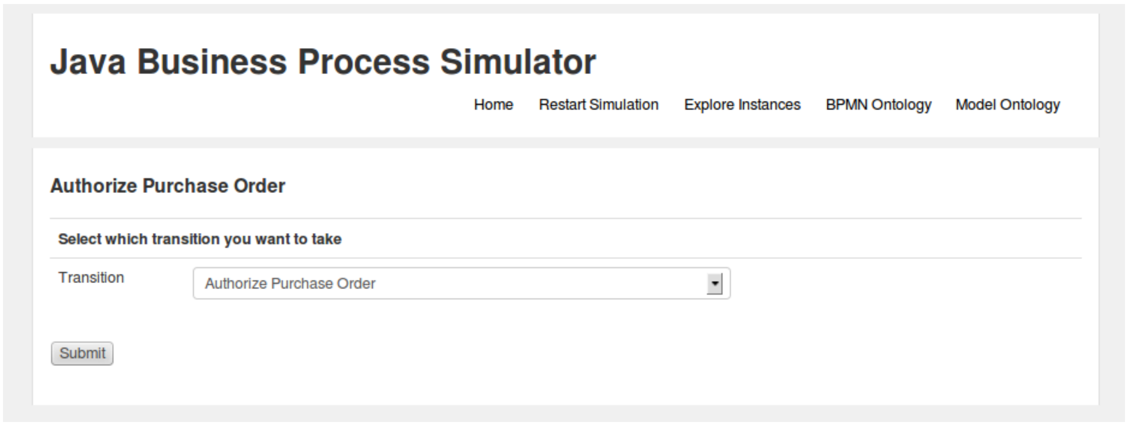
Submit

Figure 5.7: Purchase Order: Modification

transitions can be chosen.

Suppose we choose the *Reject Purchase Order* transition, we will end up in a screen like the one shown in Figure 5.7. Here we see the previously defined Purchase Order, with the same property assignment that we can modify to request authorization one more time.

Now we see a label that said that the Action Type that will be taken is an *UPDATE*, while in the Figure 5.3 the Action Type was defined as *INSERT*. The difference is that previously we where inserting an individual in the ontology and now we are updating a previously defined individual.



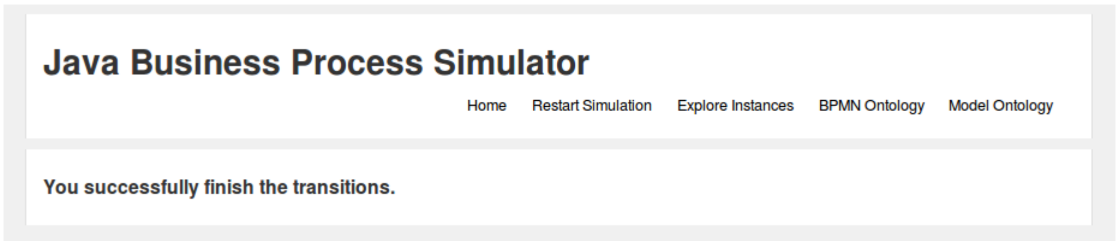
Authorize Purchase Order

Select which transition you want to take

Transition: Authorize Purchase Order

Figure 5.8: Purchase Order: Accept Purchase Order

If we submit the form we now return to the *Authorize Purchase Order* task to request for authorization of the Purchase Order, as we can see on Figure 5.8.



You successfully finish the transitions.

Figure 5.9: Purchase Order: End

Finally, if we now select the *Authorize Purchase Order* transition we arrive to the end event of the BPMN diagram, as we can see in Figure 5.9.

5.2 The Shipping Process

An example of a shipping process is shown in Figure 5.10. Basically the user first create a *Shipping List* that is a description of the products or goods that will be shipped. After the creation the approval of the *Shipping List* should be issued and the *Stock Order* can be created. Before creating it, the *Shipping List* can be modified, and finally attached to the *Stock Order*. After the approval of the *Stock Order*, the products are delivered and finally the goods arrive to the final destination.

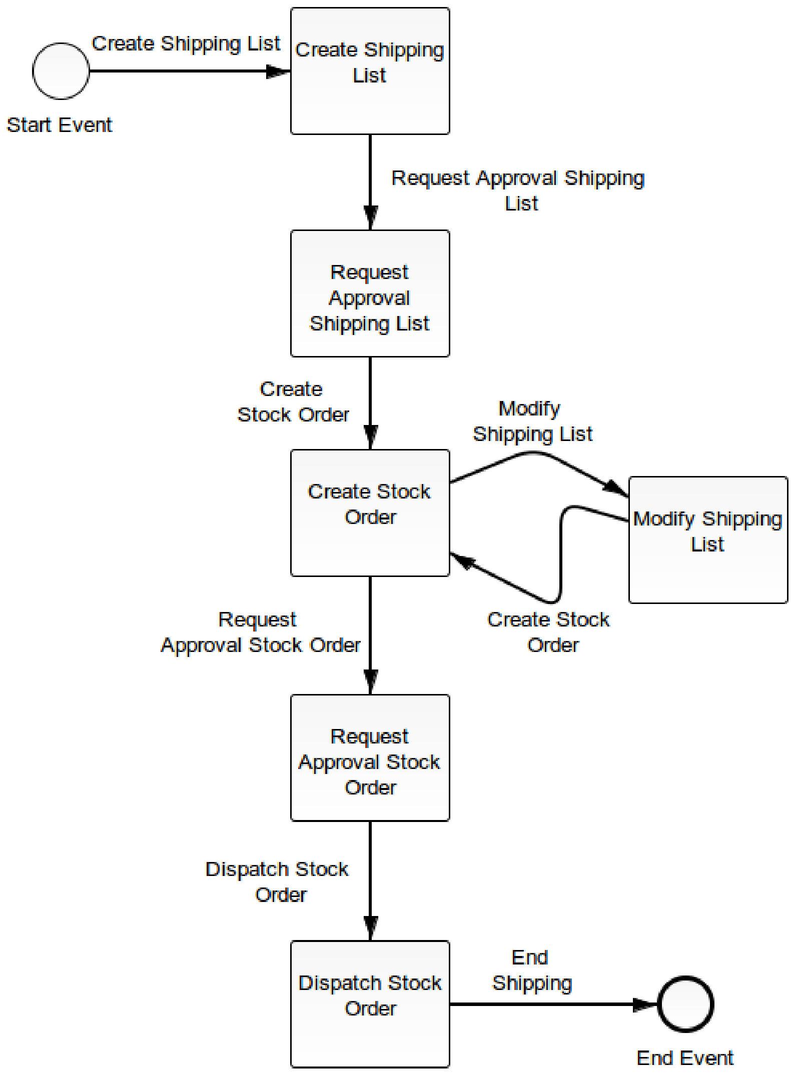


Figure 5.10: Shipping Diagram

For the underlying ontology we will use different classes. Class *ShippingList* define the Shipping List that can be created and *StockOrder* describe the corresponding Stock Orders. The class *Client* represents the Clients that we can use to associate the Shipping

Lists. Class *ShippingListState* represent the different states that a Shipping List can take. The classes that we will be using are the followings:

$$ShippingList$$

$$ShippingListState$$

$$StockOrder$$

$$Client$$

Finally we can declare in the ABox that there are three states that a Shipping List can take, this are *draft*, *approved* and *cancel*, all defined bellow:

$$ShippingListState(draft)$$

$$ShippingListState(approved)$$

$$ShippingListState(cancel)$$

Regarding properties we will define the property *stateOfShippingList* that represent the link between a shipping list and it's state. This property is a functional property with a proper domain and range defined.

$$stateOfShippingList : ShippingList \rightarrow ShippingListState$$

$$\top \ \sqsubseteq \leq \ 1 \ stateOfShippingList$$

The other important property we will be using is *orderOfShippingList*. This property represents the connection between the stock orders and the shipping orders. This property apart from being functional is also inverse functional, in a sense that stock orders and shipping orders are linked one to one, this is specified by the equation 5.13. We will use this restriction that we impose to the property to specify constrains that we will try assure in the process traversal.

$$orderOfShippingList : ShippingList \rightarrow StockOrder$$

$$\top \ \sqsubseteq \leq \ 1 \ orderOfShippingList$$

$$\top \ \sqsubseteq \leq \ 1 \ orderOfShippingList^{-} \tag{5.13}$$

Now we would like to say that if a shipping list is in the *draft* state, it cannot have a stock order associated. To define this restriction we would define a few auxiliary classes that will make the constrain easier to specify. For this we will specify the class of all shipping list that are in *draft* state and the class of all shipping list that have a stock order associated. The definition are show bellow:

$$DraftShippingList \equiv \exists\ stateOfShippingList\ .\ \{draft\} \tag{5.14}$$

$$ShippingListWithOrder \equiv \exists\ orderOfShippingList\ .\ StockOrder \tag{5.15}$$

Now we will specify that this two classes are disjoint and this is done in equation 5.16.

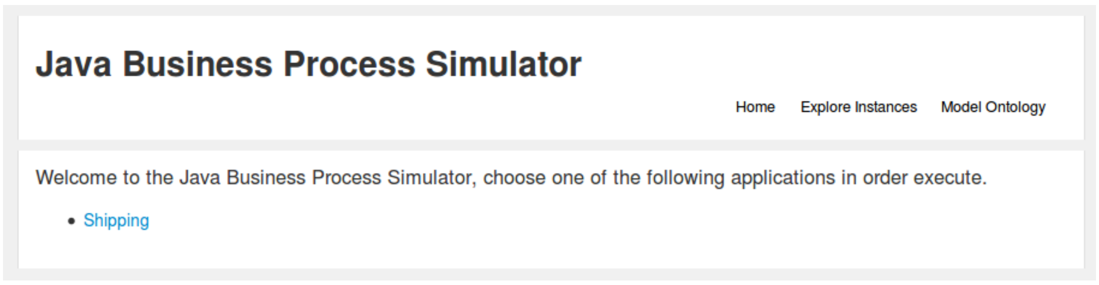
$$DraftShippingList \sqcap ShippingListWithOrder \sqsubseteq\ \bot \tag{5.16}$$

The last property we will defined is called *clientOfShipping* and captures the link between the shipping list and the client associated to it.

$$clientOfShipping : ShippingList \rightarrow Client$$

$$\top\ \sqsubseteq \leq\ 1\ clientOfShipping$$

We will not show the full JSON representation that specify the connection between the BPMN diagrams and what actions that should be taken on each BPMN task, instead we will describe it briefly. In the *Create Shipping List* task, a shipping list should be created with a selected client. In the *Request Approval Shipping List* we would like to specify the state of the shipping list that was defined in the previous task. In the *Create Stock Order* a stock order will be created and the associated shipping list should be specified, this is done using the inverse functional of $orderOfShippingList$. Finally, in the *Modify Shipping List* we can modify the state of the previously defined shipping list.

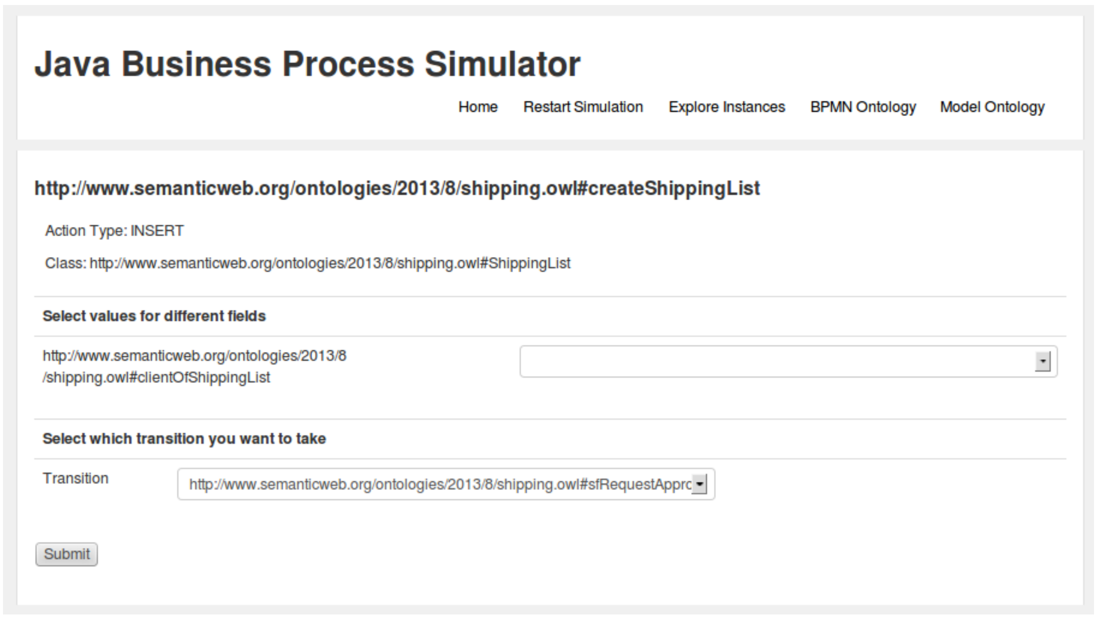


Welcome to the Java Business Process Simulator, choose one of the following applications in order execute.

- Shipping

Figure 5.11: Shipping: Start Simulation

With all this definitions we can run JBPS to traverse the BPMN diagram using the defined ontology and the task-action association to create a simulation for our shipping


Java Business Process Simulator

Home Restart Simulation Explore Instances BPMN Ontology Model Ontology

http://www.semanticweb.org/ontologies/2013/8/shipping.owl#createShippingList

Action Type: INSERT

Class: http://www.semanticweb.org/ontologies/2013/8/shipping.owl#ShippingList

Select values for different fields

http://www.semanticweb.org/ontologies/2013/8/shipping.owl#clientOfShippingList

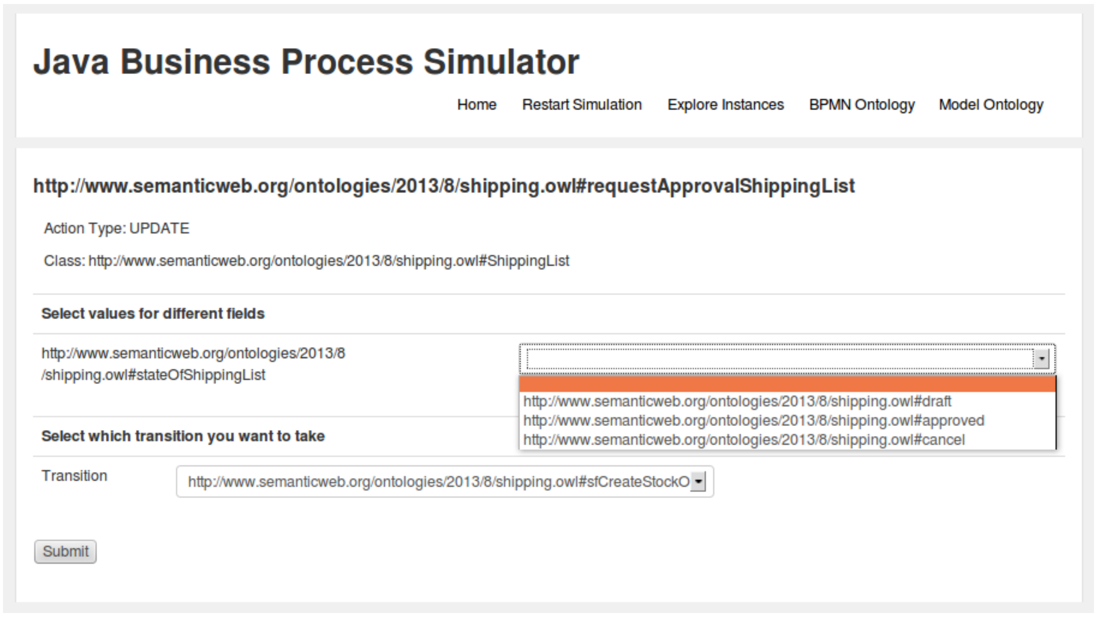
Select which transition you want to take

Transition: http://www.semanticweb.org/ontologies/2013/8/shipping.owl#sfRequestApprc

Submit

Figure 5.12: Shipping: Create Shipping List

process. Figure 5.11 shows the starting page of the JBPS simulation and in Figure 5.12 we can see page where we specify the client associated to the shipping list that is about to be created.


Java Business Process Simulator

Home Restart Simulation Explore Instances BPMN Ontology Model Ontology

http://www.semanticweb.org/ontologies/2013/8/shipping.owl#requestApprovalShippingList

Action Type: UPDATE

Class: http://www.semanticweb.org/ontologies/2013/8/shipping.owl#ShippingList

Select values for different fields

http://www.semanticweb.org/ontologies/2013/8/shipping.owl#stateOfShippingList

http://www.semanticweb.org/ontologies/2013/8/shipping.owl#draft
http://www.semanticweb.org/ontologies/2013/8/shipping.owl#approved
http://www.semanticweb.org/ontologies/2013/8/shipping.owl#cancel

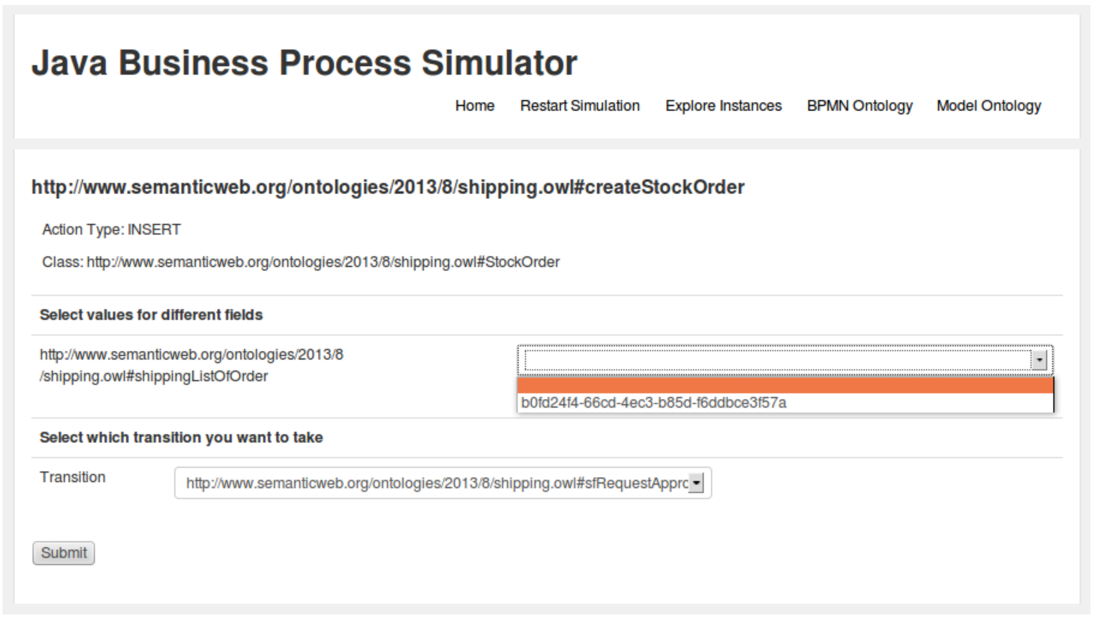
Select which transition you want to take

Transition: http://www.semanticweb.org/ontologies/2013/8/shipping.owl#sfCreateStockO

Submit

Figure 5.13: Shipping: Request Approval

Moving in the diagram we arrive to the *Request Approval Shipping List* task, where we specify the state of the newly created shipping list. Let's set the state to *draft* and continue with the BPMN diagram.


Java Business Process Simulator

Home Restart Simulation Explore Instances BPMN Ontology Model Ontology

http://www.semanticweb.org/ontologies/2013/8/shipping.owl#createStockOrder

Action Type: INSERT

Class: http://www.semanticweb.org/ontologies/2013/8/shipping.owl#StockOrder

Select values for different fields

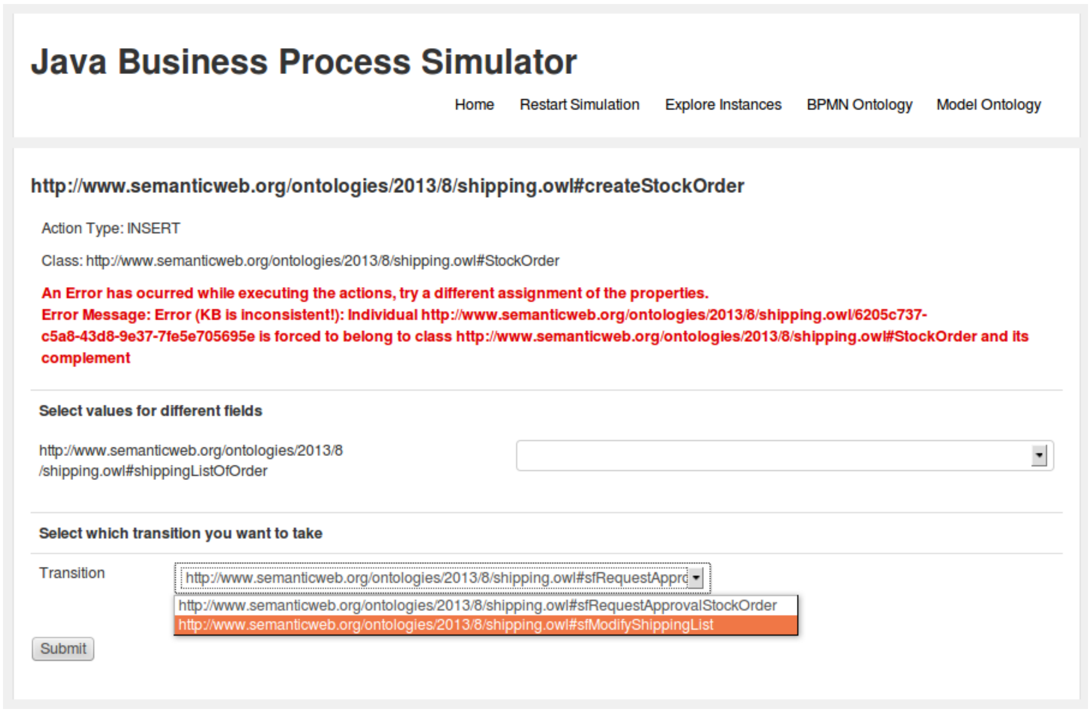
http://www.semanticweb.org/ontologies/2013/8/shipping.owl#shippingListOfOrder — b0fd24f4-66cd-4ec3-b85d-f6ddbce3f57a

Select which transition you want to take

Transition: http://www.semanticweb.org/ontologies/2013/8/shipping.owl#sfRequestApprc

Submit

Figure 5.14: Shipping: Create Stock Order


Java Business Process Simulator

Home Restart Simulation Explore Instances BPMN Ontology Model Ontology

http://www.semanticweb.org/ontologies/2013/8/shipping.owl#createStockOrder

Action Type: INSERT

Class: http://www.semanticweb.org/ontologies/2013/8/shipping.owl#StockOrder

An Error has ocurred while executing the actions, try a different assignment of the properties.
Error Message: Error (KB is inconsistent!): Individual http://www.semanticweb.org/ontologies/2013/8/shipping.owl/6205c737-c5a8-43d8-9e37-7fe5e705695e is forced to belong to class http://www.semanticweb.org/ontologies/2013/8/shipping.owl#StockOrder and its complement

Select values for different fields

http://www.semanticweb.org/ontologies/2013/8/shipping.owl#shippingListOfOrder

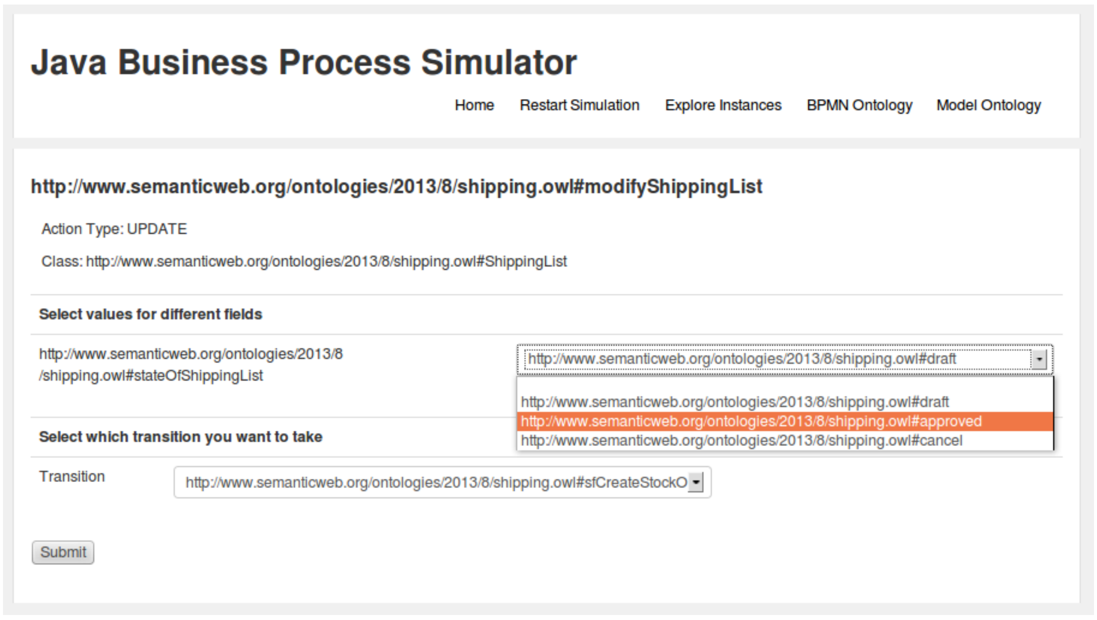
Select which transition you want to take

Transition: http://www.semanticweb.org/ontologies/2013/8/shipping.owl#sfRequestAppro
http://www.semanticweb.org/ontologies/2013/8/shipping.owl#sfRequestApprovalStockOrder
http://www.semanticweb.org/ontologies/2013/8/shipping.owl#sfModifyShippingList

Submit

Figure 5.15: Shipping: Create Stock Order constrain violation

Now we should create a stock order with a shipping list associated to it and we can only choose the newly created shipping list as we can see on Figure 5.14. If we try to create a Stock Order with this association, JBPS will fail as shown in Figure 5.15 since we are associating a stock order to a shipping list that is in state *draft* and this violates the constrain imposed with equation 5.16.


Java Business Process Simulator

Home Restart Simulation Explore Instances BPMN Ontology Model Ontology

http://www.semanticweb.org/ontologies/2013/8/shipping.owl#modifyShippingList

Action Type: UPDATE

Class: http://www.semanticweb.org/ontologies/2013/8/shipping.owl#ShippingList

Select values for different fields

http://www.semanticweb.org/ontologies/2013/8/shipping.owl#stateOfShippingList: http://www.semanticweb.org/ontologies/2013/8/shipping.owl#draft

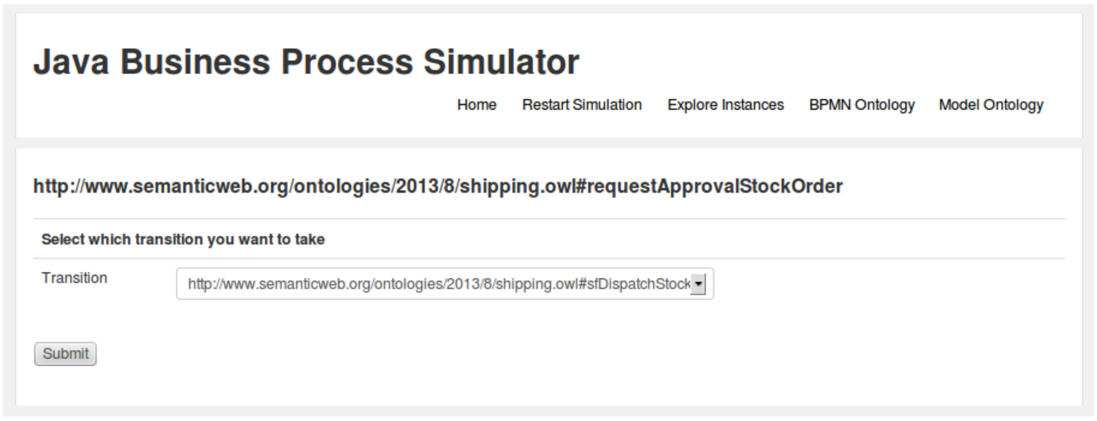
http://www.semanticweb.org/ontologies/2013/8/shipping.owl#draft
http://www.semanticweb.org/ontologies/2013/8/shipping.owl#approved
http://www.semanticweb.org/ontologies/2013/8/shipping.owl#cancel

Select which transition you want to take

Transition: http://www.semanticweb.org/ontologies/2013/8/shipping.owl#sfCreateStockO

Submit

Figure 5.16: Shipping: Modify Shipping List


Java Business Process Simulator

Home Restart Simulation Explore Instances BPMN Ontology Model Ontology

http://www.semanticweb.org/ontologies/2013/8/shipping.owl#requestApprovalStockOrder

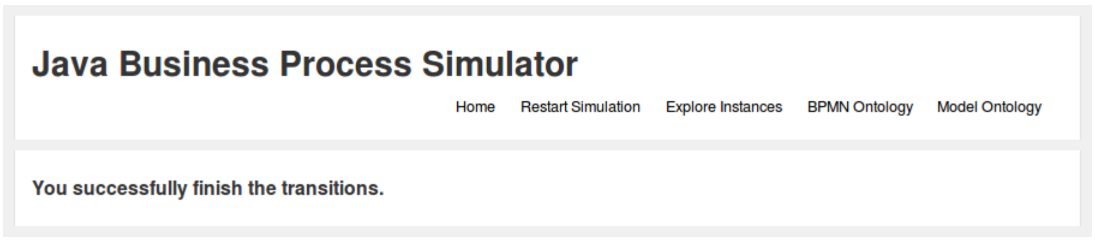
Select which transition you want to take

Transition: http://www.semanticweb.org/ontologies/2013/8/shipping.owl#sfDispatchStock

Submit

Figure 5.17: Shipping: Request Approval Stock Order

After trying to create a stock order we can take the *Modify Shipping List* transition to modify the state of the shipping list. After taking that transition we arrive to Figure 5.16 where we can modify the state to *approved*. At this point we can go back to the *Create Stock Order* task and now we can proceed normally since there is no constrain violated and the knowledge base is consistent.



You successfully finish the transitions.

Figure 5.18: Shipping: Shipping Process Ended

We can now move through the different task defined in the BPMN diagram and finally reach the end event, where the process finish, shown in figure 5.18.

Chapter 6

Conclusions

More usage of the BPMN should be done by companies to define, analyze and improve their working processes. Since the gap between process definition and process implementation is getting bigger and bigger as a consequence of the extensions of the BPMN definition and the increasing complexity of the business processes, we believe that this kind of tools, aimed to reduce that gap, will help in the fast adoption of this standards.

As shown in the study cases, a big potential exist in the simulation of complex processes in an automatic way. If the JBPS is used for testing requirements and assumptions in the process, this will drop costs in the process implementation because the cost of testing such processes is zero and it will reduce the number of iterations in the implementation as the requirements will be far more clear. If the JBPS is used as a production application to support the companies business process, the costs of implementation will *be* almost zero, since the BPMN diagrams will be available as the process are modeled, only the underlying ontology should be defined.

At this point, the fact that we make use of specific built ontologies to express constrains, not aimed in the BPMN specification, and to model the used underlying business objects should seems the right choice giving the increasing use of OWL in the industry and the interesting features it provides: expressive power and decidability. While it is true that people qualified to make a correct ontology is scarcer than people qualified to make an implementation of the business process, the definition of the ontologies is becoming easier as supporting software, like Protégé, and on-line resources are produced and adopted.

Finally it is worth noticing that there is a big amount of work that can be done in order to increase the features of the Java Business Process Simulator in order to provide a more reliable software and a more complete application. Here is a list of the most significant features that can be added, possible in future works, to the JBPS:

- The inclusion of roles and users as a way to restrict the possible actions to group of users. This will be interesting since actions sometimes can be performed only by a selected group of users and not by anyone.

- The definition of collections, in an OWL way, and possible usage of them in the underlying ontology model. Also the necessary rendering in the pages should be performed to allow the user to manipulate such collections.
- The usage of some storage facility, possible a database like ontology engine. This will allow to save the current configuration of the ontology and to make all operations done with databases, such as back-up, mirroring, data that survive application crash, etc.

Bibliography

[1] The bizagi bpm suite. `http://www.bizagi.com/`.

[2] The java business process simulator. `https://github.com/damiansoriano/jbps`.

[3] Object Management Group Inc. Bpmn 2.0 by example. `http://www.omg.org/cgi-bin/doc?dtc/10-06-02.pdf`, June 2010.

[4] Object Management Group Inc. Business process model and notation v 2.0. `http://www.omg.org/spec/BPMN/2.0/PDF`, January 2011.

[5] Peter Korju. The bpmn pizza store diagram tutorial. `http://processpedia.com.au/community/the-bpmn-pizza-store-diagram-tutorial/`, October 2012.

[6] Bijan Parsia and Evren Sirin. Pellet: An owl dl reasoner. In *In Proceedings of the International Workshop on Description Logics*, page 2003, 2004.