Abstract: This paper presents how Functional Programming (FP) helps to provide an other formal semantics (relation between the syntax and the model of computation) for Business Process Modeling (BPM); a semantics relatively different from Object Oriented semantics. More precisely, it proposes a general methodology to model business processes using mathematical functions and higher-order functions. We describe the basic part of Business Process Modeling, behavioral semantics via Petri Nets (PN) and Functional implementation of the models. Also, we will see how the business process model is translated into its equivalent form in Petri Nets and how these can be described through Functional Programming.

Keywords: Functional Programming, Semantics, Business Process Modeling, Petri Nets.

1. INTRODUCTION

The term "Business Process Modeling" was coined in the 1960s in the area of systems engineering by S. Williams and until the 1990s, it became popular. BPM is the way of describing processes within enterprises, so that a model (abstract representation which can be manipulated) can be analyzed and improved, (Aguilar-Saven, 2004). A business process is viewed as a set of related, structured activities or tasks within an organization whose objective is to produce a specific product or service. A task needs to be finished before its deadline or in a definite time to work towards goal. There are many techniques to model processes such as a flow chart, functional flow block, Unified Modeling Language (UML), Business Process Model and Notations (BPMN), etc.

Functional Programming (FP) is a style of programming in which programs are executed by evaluating expressions. Functional Programming focuses on simplicity and generality, (Hughes, 1989), functions are considered first-class, which states that they can be passed as parameters to other functions or be returned as a result of a function. FP does computation as the evaluation of mathematical functions without changing state and mutating data. In comparison to Object-Oriented Programming (OOP), order of execution is less important, functions with no side-effects, uses recursion concept to iterate collection data, supports both "Abstraction over Data" and "Abstraction over Behavior", etc.

Semantics is the study of meaning, (Lyons, 1977). In terms of programming language theory, semantics is the field related with the mathematical study of the meaning of programming languages. It describes the processes a computer follows when executing a program in that specific language. This can be viewed by describing the relationship between the input and output of a program, or an execution of how the program will execute on a certain platform, developing a model of computation. In this paper, Functional Programming has been used to propose a formal semantics of business model.

Many research papers have been presented and discuss about the concepts of Business Process Modeling with Petri Nets or Workflow nets or Object-oriented but a few are based on Functional Programming. In particular, iTask system (iTasks) is a task-oriented programming toolkit for programming workflow support application in Clean.

The paper is organized into three sections as follows: The next section discusses and introduces Business Process Modeling and Functional Programming. Our proposition is presented in Section 3. Finally, Section 4 concludes this paper by summarizing the main points introduced in the paper and its future aspects.

2. CONCEPTS AND STATE OF THE ART

2.1 Business Process Modeling

A business process is a set of tasks and resources required to achieve some services. It is also stated as a set of activities that once completed will accomplish goal. There are constraints and rules that have to be met. Basically, there are three types of business process:

(1) Management processes that govern the operations of a production system.
(2) Operational processes that constitute the core business and create the primary value stream.
(3) Supporting processes which supports the core processes.

A business process can be decomposed into several sub processes, with specific attributes. A process model is a representation (graphical or textual) of business processes represented as a set of sequential or parallel process activities combined together to achieve a common goal. Using
a model, it becomes possible to find out how the system will behave and the properties it will acquire. There are different modeling languages that are used to model business processes such as BPMN, (Herden et al., 2015), suitability of the BPMN for Business Process Modeling using workflow patterns, (Wohed et al., 2006), Modeling business process through activity diagrams patterns, (Andre et al., 2014). (Mili et al., 2010) describes the classification of business process modeling languages. The two most used graphical notation for business processes are BPMN and Unified Modeling Language Activity Diagram (UML AD) which are discussed in (Geambasu, 2012).

As an example, the Figure 1 models the behavior of the ATM machine for withdrawing money with the Business Process Modeling Language (BMPL).

The above model shows how the data flow between the different activities (represented as boxes), how conditions take places and how the flow of control takes place. The activities have some data associated with them, the data related to the current activity and its result will act as an input to the next activity. The arrow shows the transition between the two activities which involves the transfer of data and also the flow of control between activities and it also describes the change in the state of the activities. The diamond is used as a condition which has two outputs and the selection of the output is depend on the output of the previous activity which acts as an input to the condition.

The different semantics that are used to give a behavioral description of the business process model are Petri Nets or Workflow nets. A Petri Net is represented as a bipartite graph that have tuple $N = (P, T, A)$ where $P$ is the set of places, $T$ is the set of transitions and $A \subset (P \times T) \cup (T \times P)$ is the set of flow relations, (Zhang et al., 2010). They are the basic model of parallel and distributed system. They were documented by Carl Adam Petri in 1962 in his PhD thesis "Kommunikation mit Automaten". The basic idea behind it, is to draw the changes of state with transitions in a system.

Petri Net is a strong technique for modeling and analysis of the system behavior where resource sharing, concurrency and synchronization are a significant matter to take into account, (Reisig, 1985). A classical Petri Net is that which consists of place nodes holding resources, the number of resources is denoted by the number of anonymous tokens on the places, transition nodes consuming and producing resources, and arcs between the nodes specifying dependencies between transitions and places with resources on them. A transition is only enabled if all the input places have sufficient tokens, and an enabled transition can go off by consuming tokens from all the input arcs while synchronously producing tokens along each output arcs.

The Figure 2 gives an example of Petri Net which contains places ($S1$...$S7$), place contains tokens and transitions ($T1$...$T7$) that are joined by directed arcs.

A workflow net is a Petri Net that has two special nodes, the first node is known as starting node that is start place and the other node is known as final node that is end place. In a Worflow net every transition is on a path from start to end place. (van der Aalst, 1998) introduced workflow management as an application domain for Petri Nets. The Figure 3 gives an example of Workflow net.

For a basic purpose, it is sufficient to consider classical Petri Nets to model business process, (van Hee et al., 2013). But sometimes to evaluate process, the classical Petri Nets are not sufficient, that’s why the extension of Petri Nets are needed that are know as high-level Petri Nets, especially Coloured Petri Nets (CPN), (Jensen and Kristensen, 2009). In CPN, places are defined with the type that is colour set of tokens, tokens and resources are replaced by individual (or coloured) tokens defined with typed values of some inscription language, sometimes at same place multiple tokens may be found. Transitions and arcs are defined by program code that operates on the token values. Incoming arcs are annotated with variables or patterns which has a code to the transition and to the output arcs. Transition is also parameterized by the values of input tokens, can have additional binding values, can have guards (or conditions) that determine whether the given set of tokens is sufficient for firing of the transition, (Reinke, 2000).

A workflow is a representation of a sequence of activities, declared as a work of persons, of a simple or complex mechanism, or machines. A workflow can be considered a view or picturization of real work. The flow may refer refer to a document, services or a product that is being
Functional Programming

Functional Programming is a programming paradigm which models computation as the evaluation of expressions, programs are defined by a set of functions and immutable data. Functional Programming is a declarative programming model which means programming is done with expression or declaration instead of statements. Functional Programming requires functions to be treated like other values and can be passed as arguments. A function is a relation or expression that involves one or more variables, or a function \( f \) can be defined as a subset of a Cartesian product of two sets \( X \) and \( Y \), such as for all \((x, y)\) and \((x', y')\) in \( f \), if \( x = x' \) then \( y = y' \), (Coquand, 2008). Functional Programming mainly focuses on "What Information is desired that are Inputs" and "What Transformations are required that is Actual Logic".

The different languages that support Functional Programming are Ocaml, Erlang, Haskell, Scala, etc. The main features of Functional Programming languages are:

1. Higher-order functions: are those functions that take other functions as their arguments and returns a function as its result.
2. Immutable data: instead of altering the actual values, copies are created and the original is preserved.
3. Referential transparency: it states that the computation yields the same value each time they are invoked.
4. Recursion: a function which calls itself, it is the only way to iterate in FP.

The advantages of Functional Programming languages are bugs-free code, efficient parallel programming, supports nested functions, increase re-usability, better modularity, robust and reliable code, better performance. The disadvantage of Functional Programming language is requirement of lot of memory. In comparison to imperative programming it provides better support for structured programming and programs are shorter and easier to understand than their equivalent imperative programs.

We have chosen Haskell as a tool, so now we will discuss about it. Haskell is a purely (a function has no side-effects) functional programming language. The language is named after Haskell Brooks Curry, whose work in mathematical logic plays as a foundation for functional languages. In Haskell, the evaluation of a program is corresponding to deriving (Show) a function as its result. The "map" function in Haskell is an example of a higher-order function. The "map" takes arguments as a function \( f \) with a list of elements and applies that function to every element in the list and produce a new list. This is how the type signature of "map" is defined in Haskell:

\[
\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]
\]

```haskell
map :: (a -> b) -> [a] -> [b]
```

```haskell
map [ ] = [ ]
```

```haskell
map f (x : xs) = f x : map f xs
```

It says it takes an "a" and returns a "b", a list of a's and return a list of b's. An example of "map" function is:

```haskell
removeNonLowerCase :: [Char] -> [Char]
removeNonLowerCase s = [ a | a <- s , a `elem` [ 'a' .. 'z' ] ]
```

The first line describes the type signature that tells, what is the type of a variable. A type signature can be defined as:

```haskell
type = Int | type -> type | [type , type] | (type , type)
```

The type is defined in the standard library as:

```haskell
data Person = Person { firstName :: String , lastName :: String , age :: Int }
```

The first line describes the type signature that tells, what is the type of a variable. A type signature can be defined as:
map (+3) [2, 5, 9, 15] returns [5, 8, 12, 18]

The benefits of Haskell are:

(1) Code is shorter, cleaner and high maintainable.
(2) Higher-order function to reduce the amount of repetition.
(3) Requires shorter development times.
(4) List comprehension to make a list based on existing list.
(5) Less prone to errors, higher reliability.
(6) It is easy to learn and programs are often easier to understand.
(7) Lambda expression to create functions without giving them explicit names.

3. PROPOSITION

In this section we will discuss Business Process Modeling using Functional Programming. The Figure 4 shows big picture of our proposition concept.

![Business Process Modeling Diagram]

Fig. 4. Big picture of our proposition concept.

Earlier we have already discussed about the model of ATM machine for withdrawing money (Figure 1). Now, we will describe the same model in terms of Petri Net as a part of our contribution. The Figure 5 shows how the different processes of ATM model can be illustrated through Petri Net, in this we have divided the conditional part by a pair of transitions.

![Petri Net for the ATM Figure 1]

Fig. 5. Petri Net for the ATM Figure 1.

The activities can be denoted through circles and each circle represents a place which is a simply container where resources may be located. The resources can be of any kind of data, the number of resources are indicated by the number of anonymous tokens on the places. The arrow represents a transition, which takes resources from the previous place and provides it to the next place, that is the output place. A transition is only enabled when tokens are available on all input places. When a transition is fired it consumes a token from all the input places and provides tokens to each output place. We use Coloured Petri Nets to differentiate tokens when multiple tokens are available in the same place.

Till now we have discussed how to represent business processes using Petri Nets. Now, we will see how the different elements of a business model can be viewed in terms of Functional Programming. The activities that are actions for a particular purpose, information that is data related to a particular activity and links which are connection between the different activities. More precisely, we can say that activities in the model can be represented through functions, the information can be represented through data types and the transitions that are transformations of one activity to another activity that can be represented through higher-order functions or list comprehension. Table 1 shows the relation of the different elements of a model in respect of Functional Programming.

<table>
<thead>
<tr>
<th>Business Process</th>
<th>Functional Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activity</td>
<td>Function (Data -&gt; Data)</td>
</tr>
<tr>
<td>Information</td>
<td>Datatype</td>
</tr>
<tr>
<td>Link</td>
<td>Higher-order function</td>
</tr>
</tbody>
</table>

Table 1. Translation of Business Process into Function Programs.

As we will be using Haskell language as a tool for implementing our proposition, so now we will define how Haskell can be mapped with Petri Net and we will also present a general support code for the proposition. In Haskell, we have places with names, types, initial marking, transitions with names, optional conditions or guards and arrows from input places to output places. Transitions cannot be mapped to a functions directly because for a given marking a single transition might be enabled for several combination of input tokens and there can be a possibility that the mapping would give transition functions of different types but here for easy understanding we are using the same type of functions for this proposition.

The Haskell code for single step simulation with a textual trace for the Petri Net model (Figure 5) is presented in different parts with explanation as follows:

In Figure 6, the third line says that we have defined the type "Mark" which is the list of places in the net representing the type of marking in them, each place may contain multiple tokens in it. The "m0 :: Mark" describes that "m0" is a type of "Mark" and represents the initial marking on the first place. The lines 12, 13, 14 and 15 describes that we are making a synonym for an already existing data type, "Name" will have the name of the transition which is a string value, for the condition part, it will have parameter of type "Mark" and returns a result of "Boolean" type, for the action part, it will have "Mark" as a parameter and returns a result of same type and "Transition" will have three things Name, Condition and Action.
In Figure 7, "ts :: [Transition]" describes that "ts" is a type of list of "Transition". The general form of the "Transition" is defined as 
\[
(\text{Name}, \text{Condition}, \text{Action})
\]
where the "Name" represents the name of the transitions, "Condition" is represented through anonymous function which will check the input place is not empty and "Action" is defined as, the transition of "first markings" from input places to next output place and the process continues until all the markings from the list of markings on input place has been exhausted or certain conditions were not met causing the transition to stop.

In Figure 8, shows the function "printTS" which prints the name of the transitions. This function prints the transitions if they are available otherwise it will wait and check for other transitions to print.

In Figure 9, shows the function "enabledTs" which checks the available transition in the net. Depending on the availability of tokens on input place it checks if the transition is possible and if the condition is met it takes the available transitions and enables them.

In Figure 10, shows the "applyTs" function which compares the name given by the user with the name of available enabled transitions, if the name matches then the defined Action will be applied to the model or the model will end and user will have to start it again.

The graphical specification of communication structure (Nets) is represented in the form of textual specification so that the programmer can have easy access to perform experiments and data manipulation. The limitation of this code that it has single marking on the input place and every transition will perform a single action on that marking which is predefined in the code. Depending on the user, the code can be further improved by taking different markings on different places of different types like Int, Long, String, Double. And also, the code can be further modified so that the user can provide markings during runtime which can also be saved to a text file and can be used later. This code can be found at: https://github.com/SAINIAbhishek/Haskell-Program.
4. CONCLUSION AND FUTURE WORK

In this paper we have presented a new semantics for Business Process Modeling and how to model business processes using Petri Nets. We have also presented Business Process Modeling using Functional Programming for which we have used Haskell language. The code has been presented as an example to help our proposition. We have also drawn a comparison between Functional Programming and Object Oriented Programming. We have also discussed important features, advantages and disadvantages of Functional Programming and also discussed the benefits of Haskell language.

For future work we plan to develop a generator that will automatically translate the business process model into the Functional Programming concepts. We have to do more experiments on more complex models to find out its efficiency. We proposed to develop a Graphical User Interface (GUI) that can be considered to facilitate the use as a tool/simulator for translation purposes.

REFERENCES


