UNIVERSITY OF TARTU
Institute of Computer Science
Software Engineering Curriculum

Octavian Vinteler

Lightweight BPMN Execution Engine

Master's Thesis (30 ECTS)

Supervisor(s): Luciano Garcia Banuelos

Tartu 2016

Lightweight BPMN Execution Engine

Abstract:

This paper presents a BPMN execution engine which provides both model annotation and execution perspectives. With this tool, users annotate a given BPMN model with data definitions that are later used for animating the process defined in the model. During the animation, the user can enter actual data via dynamically generated forms that are attached to user tasks and message events. Data can be then processed via scripts that are specified on script tasks. Moreover, the process data is used to determine automatically the flow of execution according to the conditions specified on inclusive/exclusive OR gateways. In addition, the tool also allows processing on a distributed environment, such that multiple users can take part in the execution of a process. In sum, the tool can be described as a lightweight, self-contained system, which does not require any type of installation or configuration effort from the user, in order to start the animation of BPMN models.

Keywords:

Business Process Management, BPMN semantics, Distributed Systems

CERCS: P170 Computer Science, Numerical Analysis, Systems, Control

Kergekaaluline BPMN käitusmootor

Lühikokkuvõte:

Käeoslev diplomitöö käsitleb äriprotsesside modelleerimiseks kasutatava keele BPMN käitusmootorit, mis võimaldab mudelite annoteerimist ja täidesaatmist. Antud tööriist võimaldab kasutajal BPMN mudeli andmekirjeldustega annoteerida, mida hiljem kasutatakse määratletud protsessi animeerimiseks. Tegumitele ja teadete sündmustele on lisatud dünaamiliselt genereeritud vormid, mille kaudu on kasutajal võimalik animeerimise käigus lisada vajalikud andmed ja seejärel need määratletud skripti tegumitega töödelda. Protsessi andmeid kasutatakse vastavalt loogiliste OR-lüüside tingimustele täitmisvoo automaatseks määramiseks. Lisaks võimaldab tööriist töötlemist hajutatud keskkonnas nii on võimalik mitmel kasutajal samaaaegselt täitmisprotsessist osa võtta. Kokkuvõtvalt võib tööriista kirjeldada BPMN mudelite animeerimiseks mõeldud kergekaalulise iseseisva süsteemina, mis ei nõua kasutajalt installeerimist ega konfigureerimist.

Võtmesõnad:

Äriprotsesside juhtimine, BPMN semantika, Hajussüsteemid

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine

Table of Contents

1 Introduction

1.1 Context and Motivation

1.1.1 Business Process Management

In order to define Business Process Management (henceforth abbreviated BPM) it is important to underline that this concept has been defined in various ways by different stakeholders involved, such as: vendors, analysts, researchers, authors, or customers. In [1], the authors argue that BPM is a management discipline which is concerned only with business process management and as a mean to improve the overall performance of an organization, and is not a concept which is strongly connected to a specific technology or to technology use, in general. However, under certain circumstances technology can be a valuable way to improve the overall process quality. In fact, nowadays, most processes contain activities which are partially or completely done in an automated manner.

The idea behind BPM is that each product or service offered by a company is the result of a number of distinct activities. A set of activities which is performed in a particular order in order to achieve a specific goal which is of value to the customer or the organization itself is called a process. In general, each organization performs regularly a number of processes, each process dealing with a different aspect of the organization's activity. However, it is worth mentioning that a business process can cross the boundaries of a specific organization and can interact with a business process of another organization [3].

In [2], the authors define BPM as "the art and science of overseeing how work is performed in an organization". They identify the goals of BPM as ensuring the correct and expected behaviour of a process with the continuous intention of taking advantage of improvement opportunities. They continue by defining some of the most common ways to improve productivity, such as: reducing costs, reducing execution times, or reducing error rates. One important observation made by the authors is that BPM is not concerned with improving the way in which an individual activity is performed, but rather with improving the work performed at a process level.

In [1], the authors underline an important aspect of BPM, namely that the process improvement task within an organization is a continuous one and cannot be considered done at one given moment in time. Rather the organization has to continuously look for new ways to improve their processes through time in order to add more value to the customer or to the organization itself. For this reason, BPM should be seen as a permanent part of the organization's activity, rather than a one-time improvement effort.

One important feature of BPM is that the activity of the organization is seen in a process-oriented approach, rather than the traditional function-oriented approach. In the function-oriented approach, the entire organization is seen as a set of departments, each of them concerned with a different field of activity within the organization. However, more often than not, one process of the organization is spread across multiple such functional departments. For this reason, in BPM, the activity of the entire organization is rather divided in a set of processes, than based on the departments of activity [4]. The main advantage of this way of seeing the organization is that processes are naturally producing business value, and therefore it is very easy to determine the processes which require improvement in order to achieve the business goals of the organization.

Nowadays, BPM represents a domain which is of interest to two very different communities: business administration and computer science, which usually have very different goals and educational backgrounds. In addition, the reasons for which this subject interest these two communities are also rather different. In general, the business administration individuals are looking for ways in which they can improve the operations of a specific company, unlike the software community, whose goal is to provide reliable and scalable software systems which can be used when performing business process management activities [3].

1.1.2 Business Process Model and Notations

One of the most important concepts when talking about BPM is the necessity to be able to represent not only the activities involved in any process, but also the relationships between them, or the order in which these activities have to be performed. Due to the fact that such concepts are harder to express using simple plain text, in general, graphical representations are preferred for describing a process and its activities. Such graphical notations offer a clear understanding on the state in which a process is at a given point in time, are able to express the order of the activities in the process, and provide an easy to follow representation on the dependencies and constraints of the activities involved in the process. The role of a graphical notation is to define all the graphical elements which can be used to describe the process and to define the possible combinations allowed by the semantics of each such element, therefore providing a common standard for representing business processes [5].

Nowadays, there are multiple such graphical notations which can be used with BPM. However, it is worth mentioning that, in general, these notations are quite similar, and they usually differ only in the way they represent different elements of the process. From all the different graphical notations used with BPM, the one which is the most widely spread is called: Business Process Model and Notations (henceforth abbreviated BPMN) [3]. The main advantage of BPMN is that it provides a graphical representation in the form of a diagram, named Business Process Diagram which can be easily understood by all the stakeholders involved in BPM, and it offers a common language for business analysts, technical developers, and for the people who are responsible for implementing and managing the process [6]. Another characteristic of BPMN is that it is really suitable for web services, due to the fact that it has been designed in such a manner so that it is easy to integrate in such systems [7].

A BPMN diagram is a directed graph, in which nodes represent activities, events, or decision points, while edges determine the relationships between the previously mentioned elements and the order in which such activities have to be executed. In BPMN, different types of elements are represented by using different geometrical forms, such as rectangles, circles, or diamonds, in order to provide an easy way to distinguish between the different classes of elements. By using this notation, it is also possible to represent more complex real-life scenarios, such as expressing that two activities can be executed in the same time, or that one activity can only be executed after some prerequisite activities have been performed.

The typical way in which a process can be improved or redesigned involves two different phases. First the As-is scenario can be modelled as a BPMN diagram. Once this scenario is represented in a visual manner, the stakeholders can start thinking of different ways in which the process can be improved. The result of these remodelling is called the to-be scenario, and defines the way in which the business process has to be modified in order to be more productive.

In general, BPMN diagrams can be seen as a blueprint for the business process, in the sense that it can be used for organizing the work involved in the process [3]. Such diagrams can be used with different levels of details, starting from the overview of a large business process, up to the details of a very complex sub-process which can be modelled in a separate diagram. It is important to underline the difference between a business process model and a business process instance. Therefore, such a business process diagram represents the business process model or the blueprint, as described earlier. However, a business process instance refers to a concrete execution of the process. In other words, the same process can be executed a number of times within an organization. Each such execution, represents a different business process instance, and may be different from the other instances.

It is important to notice that BPMN is only concerned with those aspects of the business which are connected to the business processes, and does not cover things such as: organizational structure, data and information models, business strategy, or business rules. Such aspects of the organization's business are outside the scope of BPMN [4].

1.2 Problem Statement

In order to understand, improve or design a business process several different techniques can be used. One of the most common such method used, is to model the process in BPMN and afterwards to animate the execution of the process using a business process modelling tool. BPMN Animations represent one of the most convenient and efficient ways to understand, evaluate, or redesign processes, due to the fact that it provides a proper way to evaluate the performance of alternative designs [8]. The main role of animations is to allow the stakeholders to see the potential outcome of the process before implementing it in a real life environment, by allowing them to anticipate the probable results.

Each animation is based on a designed BPMN model which defines the activities and decision points of the animated process. Basically, a BPMN animation would allow the user to animate the execution of the business process, in a step-by-step manner, focusing on process execution flow, while being able to take execution decisions, such as choosing the path to follow after a decision point. This technique is very useful in understanding an existing process or discovering the anomalies, inconsistencies, inefficiencies or improvement opportunities of a newly designed process. Using such BPMN animations, one is able to determine really fast the impact of a potential change to the business process, by comparing the newly modified process in terms of performance against the old process, eliminating the need to assume the risks of implementing the change in the real life process [9].

Nowadays animations have become an important part of the BPM, and business analysts use them in order to evaluate the impact of any modification or redesign to the overall business process. Due to the popularity gained by the animations, most BPM commercial tools have included the possibility to animate the execution of the modelled processes. By using animations, business analysts can reduce considerably the risks associated to changes and improve the precision of their expectations towards the impact of any process design decision. In addition, animations also prove useful when multiple what-if scenarios are considered, and a decision has to be taken regarding the best option available, given the available resources of the organization.

1.3 Objectives

The main idea of this work is to provide the business analysts with a tool that allows them to animate their business processes, and to attach specific forms to each element defined in the process in order to determine the exact execution flow and behaviour. Using this tool, the business analyst, would be able to start an animated execution of the modeled business process, and by clicking on any active process element, the application will display a form which would allow the user to set the values for specific domain variables or to decide the path to follow after a decision point in the execution flow. In this manner, the analyst would be able to present and explain how the instance case could evolve during an actual execution, to the other stakeholders, based on the decisions specific to the process. Therefore, the application is meant to combine the elements of the classical BPMN process animation with the possibility to define forms for each process element, forms which would be used to determine the behaviour of the animated process in an automatic manner, with as little manual intervention in the process execution from the user, as possible.

The objectives of the work can be divided in a few distinct categories, which will be presented in the rest of this section. The first major objective in the development of the tool, is to assure the correct behaviour of each BPMN element which would be used in designing the processes through BPMN diagrams. Such elements include tasks, gateways, events, sequence flows and sub-processes. It is of outmost importance to make sure that all the BPMN elements behave as expected during the animations in order to guarantee the correct execution of the process. In addition to individual elements, some complex element combinations, such as boundary events attached to other elements, have to be handled as well. The correct behaviour of the animated process flows is of capital importance, since all the other features of the application are dependent on the ability to successfully animate BPMN processes and also because the application is designed to be an animation tool in the first place. Special attention is required for gateways, where several behavioral decisions are required in order to determine the way in which such elements behave during the animation, due to the fact that in some cases there is no generally accepted behaviour.

The second objective of the application, is to provide the ability to define and interact with forms attached to BPMN elements. This piece of functionality is required for the business analyst to be able to control and determine the process case evolution flow. Many of the BPMN elements require some form attached, however these forms have a rather different purpose, depending on the type of the element they are attached to. Each user task defined in the BPMN diagram would present the user a form, when it is executed. In this form, the user would be able to set the values for different domain variables, which would be used later in the process animation for taking decisions (i.e. choosing the path to be followed after a XOR split gate). In addition, the user can use a different form to define which such domain variables are modifiable by each task. Apart from the regular tasks, the application allows the user to define script tasks, where the user can enter a script to be executed during the execution of the task. Message events would present a form in which the user can define the message afferent to each event. In order for the application to take automated decisions after an exclusive split gateway, one form is required for defining the conditions, base on which, the path that follows the gateway is chosen. These conditions would be evaluated and, if possible, one outgoing path will be chosen if its condition can be evaluated to a value of "true".

The last major objective of the application is to provide a distributed environment for the animations, in which different users would interpret different business roles. In this way, they would be able to interact with the parts of the process which are attributed to their role, without being able to see the decisions taken by the other users, only the overall progress of the process instance. Such roles can represent different stakeholders involved in the process, or they can simulate different departments of the same organization. One of the main benefits of such an approach is that sensible information can be provided only to the users which are entitled to work with it, while being concealed from all the others stakeholders involved. Another important aspect is that such animations can be conducted remotely by different people involved in the business, each taking part according to its role within the organization.

By combining these three major features into one application, the result is a distributed BPMN Simulator which allows the user to define the business domain of the process, and to interact with it during its execution. The animation would become more realistic due to the fact that real-time decisions can be taken which would affect the latter evolution of the process instance being executed. In addition, during the animation, certain information which is relevant in the real-life process can be retained in the forms attached to different elements, while the business purpose of each activity becomes well defined based on the information which can be manipulated while executing it.

1.4 Structure of the Document

In this section the structure of the remaining chapters of this work and their sections will be presented. In the second chapter the Background information will be introduced. In the first section, 2.1, the BPM Lifecycle will be presented, followed by section 2.2, in which some historical facts about BPMN are introduced. Section 2.3 dwells upon the notations of BPMN, while in section 2.4, we will focus on presenting some similar implementations and approaches. The requirements of the application will be defined in section 2.5, while the last section of this chapter, 2.6 will be reserved for a brief discussion about the background information.

Chapter 3 will be devoted to the theoretical principles and the implementation of the tool, starting by presenting the lifecycle of the animation in section 3.1. In the following section, 3.2, the semantics of BPMN used in the application will be introduced, while section 3.3 will focus on the architecture of the implementation. Again, the last section, 3.4, will be devoted to a discussion concerning all the information presented in this chapter.

In chapter 4, the obtained results will be introduced, and a case study analyzed in order to present a concrete example of how the obtained tool works. Therefore, in section 4.1, all the achievements and results of the implementation will be presented in details, while section 4.2 will be reserved for the case study.

The final chapter of this document (chapter 5), will present some conclusions related to the overall work in the first section (5.1), while section 5.2 defines some ideas as to the potential future development of the tool.

2 Background

2.1 BPM Lifecycle

The goal of this section is to provide a brief introduction into the main stages of implementing BPM in an organization, named BPM lifecycle. The different stages are presented independently, describing their utility and the expected goals or artefacts. In addition, the dependencies among different phases in the lifecycle is also introduced.

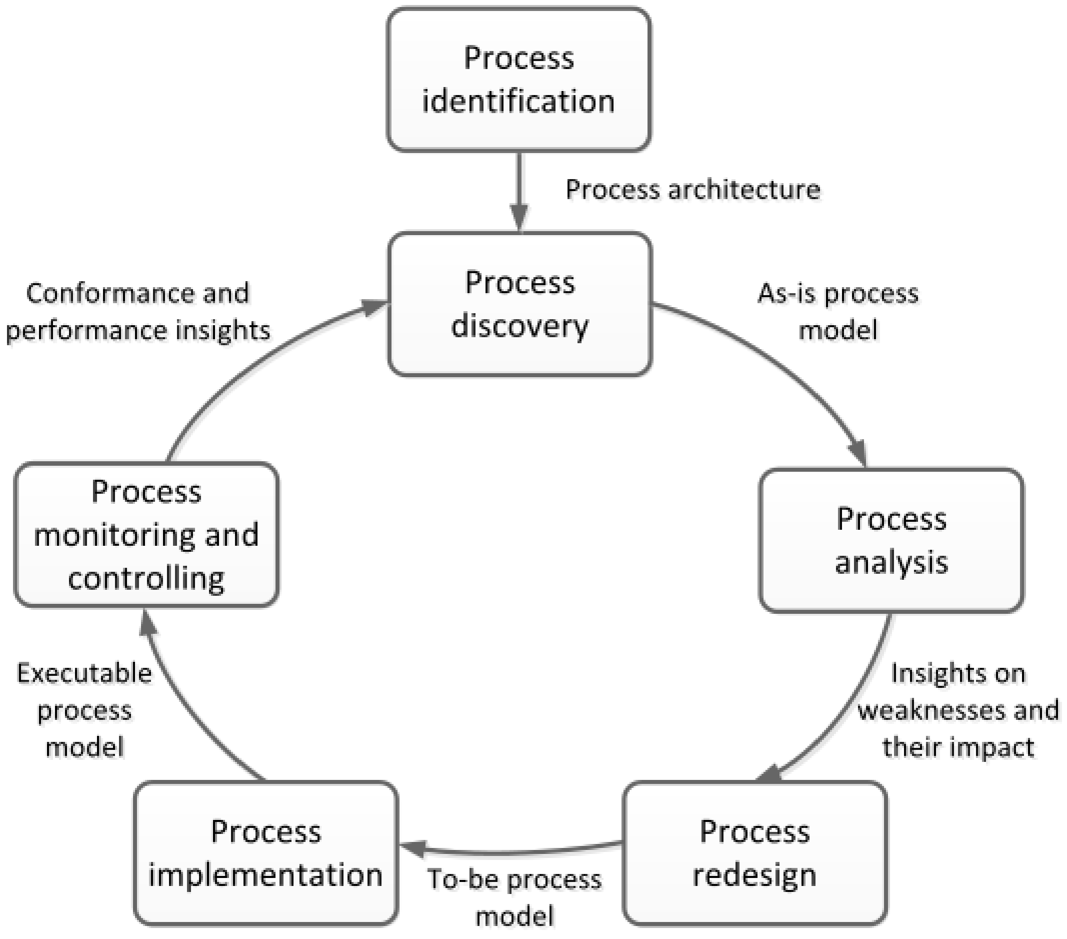


Figure 1: BPM Lifecycle - taken from [2]

The first challenge when implementing BPM is to be able to define all the business processes in the organization, to delimit the scope of each identified process, to determine which of these processes are good candidates for process improvement, and to determine the relationships between them. This first phase of BPM is called **Process Identification** and usually the result of this phase is the "process architecture", which consists of a set, containing all processes, and the relationships identified between them. One important aspect of this phase is to be able to determine the way in which the quality value of the processes can be measured. Such a measurement method is called process performance metrics. Typical performance metrics can be the total cost, the cycle time, or the error rate, which is the total percentage when an instance of the process ends up with a negative result [2].

The next step in BPM, after Process Identification is to understand in details the business processes. This phase is called **Process Discovery**, and usually the result of this phase is a collection of as-is process models. These models represent the way in which the work involved in each process is executed at the moment and they are used in order to provide a common language between the different stakeholders for understanding the existing processes within the organization. For this reason, it is very important that such models are easy to understand by all the people involved in the process. Although such models can be expressed as textual descriptions, such plain texts are usually rather ambiguous as they do not provide all the necessary information, or the information can be interpreted differently by different stakeholders. For this reason, process models are usually expressed as diagrams. The most basic notation for defining process models is called flowcharts. This type of notation works with three types of elements: activity nodes, event nodes, and control nodes. More advanced notations, based on flowcharts are UML activity diagrams, data flow diagrams, or IDEF3. However, the most widely used standard for processing modelling is BPMN [2]. It would be also very interesting to emphasize the importance that animations play during this phase in validating the model created in order to represent the as-is scenario [3].

After the process is modelled and understood by all the stakeholders it is important to identify the issues present in the current process. Issues may be of multiple types, such as long waiting time, high amount of rework that has to be done, or general issues that determine a negative outcome for the process instance. This stage, when issues are defined and the opportunities for improvement are identified is called **Process Analysis** [2].

Following-up the identification of the process flow, the next phase is called **Process Redesign**. During this stage, the identified issues are addressed, by modifying the process in order to eliminate or reduce their impact upon the process quality. It is important to remember that a modification introduced in order to correct one identified problem may introduce others issues into the process if not analyzed properly. Another aspect which needs to be kept in mind is that in some cases the proposed changes may prove to be too costly to be justified, or they may imply modification in other organizations as well, not only in the one which owns the process [2].

After the changing proposals for addressing the identified issues are made, a new version of the process has to be modeled which includes all the accepted changes in the process model. This new model is called the to-be process and reflects the way in which the process will be executed after the proposed modifications are in place. Sometimes there may be multiple redesign variants available for one process. In this case, each such option has to be analyzed and compared to the others in order to make sure that the best option is chosen for redesigning the process. The main output artefact of this phase is the to-be model, which will represent the start base for the next stage of BPMN.

During the **Process Implementation** stage, the changes included in the to-be model are implemented and when necessary the IT systems of the organization are modified in order to be able to accommodate these changes. Sometimes, during this phase it is also required to train the people involved in the process so that they can perform the activities of the newly designed process in a proper manner. Usually, these stage of BPMN contains two distinct activities: organizational change management and process automation. The former refers to the activities which need to be carried out in order to change the process, as described in the to-be model. Such activities may be: explaining the changes and their reason to the people involved in the process, so that they can understand how and why is it necessary to change the way they perform their work, developing a change management plan which may include transitional steps and dates for implementing the changes, or training users in the new way of working. Process automation is concerned with implementing or configuring the IT systems required in order to be able to support the to-be process [2].

The last phase in the cycle is called **Process Monitoring and Controlling**. During this phase the new process is monitored in order to confirm that the process works as expected. In case the process does not perform in conformance with the expectations some adjustments can be done in order to correct it. It is important to emphasize that this stage does not represent the end of BPM activities. BPM has to be seen as a continuous effort, therefore the presented phases have to be executed in a cycle as seen in figure 1, because if a process in not monitored and improved in a continuous manner, eventually it will become subject to degradation.

2.2 Historical Facts about BPMN

The first version of BPMN, named BPMN 1.0 was released in 2004 by Business Process Management Initiative (henceforth abbreviated BPMI) and in time, the BPMN notation managed to impose itself as the de-facto standard for modelling business processes [4]. In the beginning, the BPMN language was developed in order to provide notation for another standard developed by BPMI: Business Process Modeling Language (BPML), which was an XML-based standard used in BPM. BPMN was adopted by the Object Management Group (henceforth abbreviated OMG), which currently maintains and develops the standard. After its creation, the standard evolved into an independent and more general modelling notation, which benefited from a continuous development by OMG. In time, the BPML standard came to be replaced by a number of different standards, with the graphical notation part of BPML being substituted by the BPMN notation. In 2005 BPMI and OMG merged, and in 2007, BPMN 1.1 was released, which in fact represented only a minor update to the notation. However, a major step forward was made with the release of BPMN 1.2 in 2009. This version of the BPMN standard was the first to achieve widespread popularity in the world of BPM. Even nowadays many available BPM tools are based on this version of BPMN and have not yet migrated to the latest version. In early 2011, the current version of BPMN was released, named BPMN 2.0, which introduced significant changes compared to the previous versions [10].

BPMN 2.0 also introduced a standard XML serialization format, which is important because it promotes the exchange of models between tools [10]. This addition is also essential due to the fact that it provides a bridge for the gap between the activities of process modelling and process execution [11]. In addition, there are several other areas in which the 2.0 version of the standard extended the scope of the notation: formalization of the semantics for all BPMN elements, the possibility to define extensibility mechanisms for model and graphical extensions, event composition and correlation, extensions of the definition of human interactions, and the definition of a conversation view for a collaboration diagram [12].

Nowadays, many BPM tools provide conformance with BPMN 2.0, and the notation is widely seen as the standard in place for process modelling, due to the fact that it can be considered a collection of best practices used by other such notations. In this sense, we can identify a set of ancestors of BPMN, from where the notation has taken various concepts and ideas. Such notations are: graph-based process modelling languages and Petri-net based modelling languages, such as: UML activity diagrams, or event-driven process chains [3]. In general, each such model focuses on a particular level of abstraction, ranging from a general business view level, to a more in-details technical view. However, BPMN aims at accommodating a large variety of abstraction levels, in order to be usable by a wide range of stakeholders, starting from business analysts, up to technical developers. By supporting these different abstraction levels, BPMN can be used successfully during different phases of the BPM lifecycle, such as: business process design, or process implementation [3].

In order to identify the degree to which a specific BPMN software tool provides conformance with the BPMN notation, three different conformance classes have been introduced. The first conformance class is named: Process Modelling Conformance and it assures that the tool includes the BPMN core elements, and a specific number of diagram types, such as: process, collaboration, and conversion diagrams. The second conformance class is called: Process Execution Conformance, and it is achieved by a software tool by supporting the operational semantics of BPMN. Choreography Modeling Conformance, is the last conformance class and it is achieved by providing support for the BPMN core elements, collaboration, and choreography diagrams [3].

2.3 BPMN Elements

Due to the fact that it is designed to be able to represent even the most complex business processes, BPMN is a very rich language, having more than 100 different graphical elements [2]. However, not all symbols are encountered with the same frequency, mainly because a relatively small number of symbols, the so called core set, can be used in order to represent most business processes. In this section, the most common elements will be introduced and grouped in categories. Along with the visual representation of each element, a brief description will be offered, explaining the purpose of the element and the manner in which it can be used.

The BPMN elements can be divided in 5 different groups, each groups encapsulating elements with a common behaviour and purpose. These categories are: Flow Objects, Data Objects, Connecting Objects, Swimlanes, and Artifacts [4].

2.3.1 Flow Objects

This category contains some of the most important BPMN symbols used for process modeling. Each business process contains: events, activities, and, most likely, decision points, for representing the usual steps required for completing the process. Therefore, we can say that the Flow Objects are used to describe the individual executional or decisional points in a process.

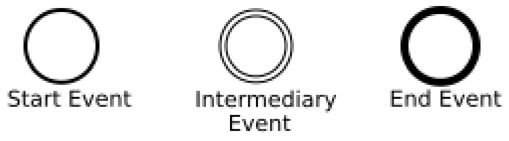


Figure 2: Event types

The first category of Flow Objects, are the **Events**. Events represent something that happens instantly, and do not require an amount of work in order to be completed. There are three different types of events: Start Events, Intermediate Events, and End Events. **Start Events** represent the events which start an instance of the process, such as receiving an application or an order. Start events are represented with thin border circle, as shown in Figure 2. On the other hand, **End Events** are used to signal the end of the process instance, such as completing an order or delivering an e-mail. End events are represented as circles with thick border as can be seen in the same figure. **Intermediate Events**, are events which occur during the execution of the process, usually blocking the process execution until the event happens. Intermediate events are represented as double border circle (figure 2).

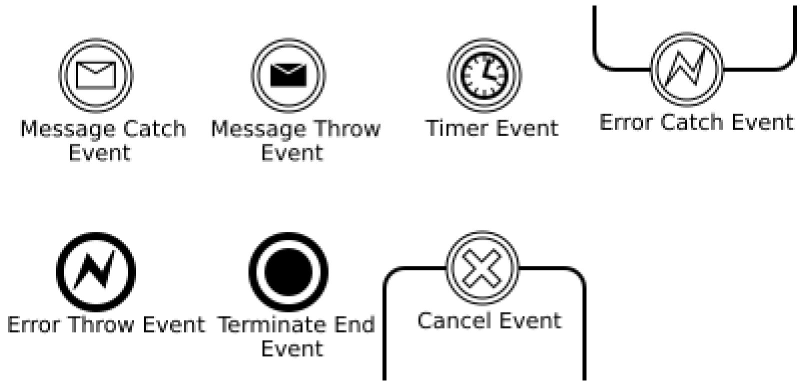


Figure 3: Different Types of Events

In addition to this three main categories, events can be further grouped, depending on their trigger or result type. The most used types of events are Message Events, Timer Events, and Error Events. **Message Events** are represented with an envelope symbol inside the circle, as can be seen in Figure 3, and the presence of one of these events denotes that during the process a message is expected from or sent to another entity. Message events can be present in the beginning of the process, as start events, during the process, as intermediate events, or at the end, as end events. When the meaning of the event is that a message is expected, the envelop is white, while if the message is sent, the envelope is coloured in black. Another type of events which is commonly used in BPMN are the **Timer Events**, which can also be seen in figure 3. They are represented by a clock symbol inside the event circle. Depending on the context of the process, a timer event can denote that something happens at a specific point in time, or after a specific time interval [2].

BPMN defines other types of events used to represent a deviation of the process from its normal course. The first such event is the **End Terminate Event**, represented as an end event with a full circle inside (figure 3), and it is used to signal an improper termination of the process. In order to handle an exception in one activity of the process, an **Error Event** can be used to signal the error and to bring the process into a consistent state again. These error events are represented with a lightning symbol inside the event circle (see Figure 3). If the event throws an error, the lightning symbol is black, while if the event is responsible for catching an error, the lightning symbol will be white. In case an activity has to be cancelled, the **Cancel Event** can be attached to this activity in order to determine an external cancellation request received during the time this activity was executed. Such an event is represented with an “X” inside the event circle (figure 3).

In contrast to events, **Activities** represent units of work which require a given amount of time to be completed and are executed inside the organization with full control over their execution. In BPMN, activities are represented with rounded-corners rectangles. However, as in the case of events, activities can be of multiple types. The most used types of activities in BPMN modelling are Tasks and Sub-processes.

Tasks represent atomic activities, which cannot be further broken down in smaller units of work. An example of the visual representation of a task can be seen in figure 5. Depending on the way they are executed, tasks can be of multiple types. The **Automated Task** is executed automatically by the software system or by an external service, without requiring human participation. On the other hand, **Manual Tasks** are processed by human participants alone (e.g. call a customer by phone). **User Tasks** are executed by the human participants that interact with the underlying information system [2]. In addition to the manner in which they are executed, the type of the task can also denote the behaviour of the task during the execution. For example, a **Script Task** represents the execution of an embedded script at a given point in the process. The type of the task is denoted by a small symbol in the upper left part of the rectangle. Examples of the described tasks can be seen in figure 4.

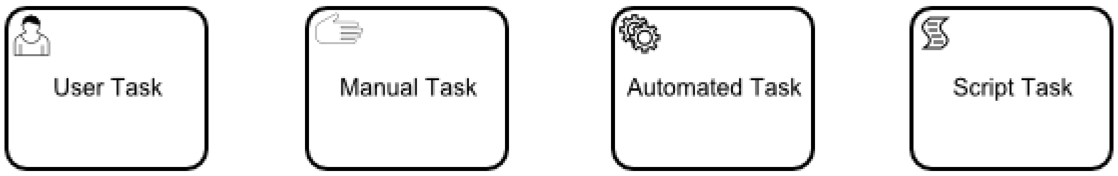


Figure 4: Task Types

Sub-processes represent compound activities, which can be further decomposed into smaller units of work. Usually, a sub-process contains a group of related activities that together would produce something of value to the process [2]. Sub-processes can be distinguished from tasks, by a plus (+) sign placed in the lower part of the rectangle. Sub-processes can be defined on multiple levels, with one sub-process, containing other sub-processes inside. The visual representation of a collapsed sub-process can be seen in figure 5.

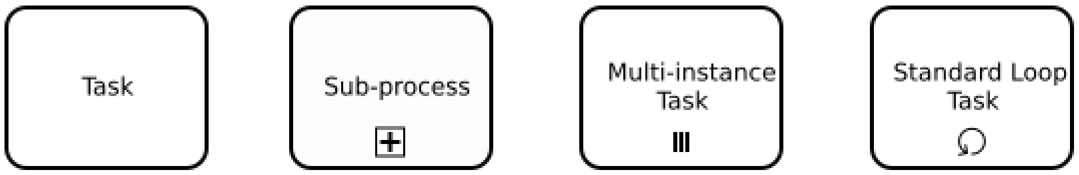


Figure 5: Activities

In addition to the regular activities already presented, in BPMN, we can also define special types of activities in order to represent particular situations. When an activity has to be executed multiple times in parallel for different entities or data items we can use the **Multi-instance Activity**. This activity is represented with three parallel vertical lines in the lower part of the rectangle (figure 5). Another very useful type of activity is the **Standard Loop Activity**, which represents a type of activity that can be executed in a sequential loop for a specific number of times [2]. This type of activity is represented using a circle-shape arrow in the lower part of the rectangle (figure 5).

The last important flow object type is represented by the **Gateways**. Gateways are used when the events and the activities in a process are not executed in a sequential manner, and the path of the sequence flow needs to be controlled. Therefore, gateways are used for branching, merging, splitting and joining the process execution paths [4]. Based on the number of incoming connection from BPMN elements and the number of outgoing ones, the gateways can be of two types: split gateways, where the process flow diverges, and join gateways, where the process flow converges [2]. Gateways are represented with different symbols inside a diamond (figure 6).

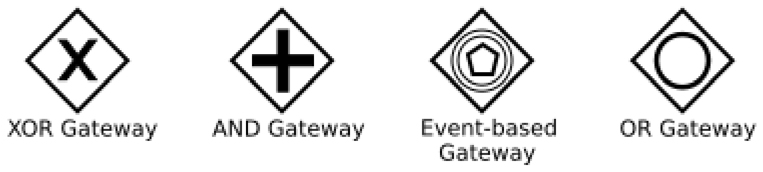


Figure 6: BPMN Gateways

The **XOR Split Gateway** is responsible for selecting one path out of multiple alternatives. The execution flow paths are re-merged latter in the process using an **XOR Join Gateway**, which will become active as soon as it receives a token on one of its incoming connections. An XOR gateway is represented with an “X” symbol inside the diamond as can be seen in figure 6.

The situation when two or more activities can be executed in any order or at the same time can be modelled with an **AND Split Gateway**. The behaviour of this gateway is that all the outgoing branches will be executed sooner or later, but the order in which they are executed is not important. After all the branches are executed, an **AND Join Gateway** can be used to synchronize the execution. The AND gateway is represented with a “+” symbol inside the gateway diamond (figure 6).

One special type of exclusive decision elements is represented by the **Event Based Split Gateway**. The behaviour of this gateway is identical with the regular XOR split gateway, with the exception that after such an Event Based Split Gateway the next elements have to be events. Basically, the selected path will be determined by the first incoming event from the set. Unlike the previously presented XOR split gateway, which models an internal choice, this type of gateway models a type of choice which is not determined within the organization, but rather by the process environment [2]. This type of gateway can be seen in figure 6, and is represented by a pentagon within two circles.

The last important gateway type is represented by the **OR Split Gateway** and, respectively, the **OR Join Gateway**. The OR Split Gateway is used to model the situation when one or more outgoing branches can be executed. Therefore, the different outgoing branches are not mutually exclusive, as in the case of an XOR Split, but they do not necessarily need to be executed all either, as in the case of an AND Split. The OR Join Gateway is used for merging the execution branches after an OR Split, and it will wait for all the active branches to be executed before it will become active [2]. The OR Gateway is represented by a circle inside the gateway diamond (figure 6).

2.3.2 Other Types of BPMN Elements

The most basic type of **Connecting Object** in BPMN is called **Sequence Flow** (figure 7), and it is used to connect activities, events, and gateways with the aim of determining the order in which the flow objects are executed within the process. Depending on the situation, a sequence flow may have a condition attached, usually after a XOR gateway. Such a condition is used to determine the path that is to be followed (the one for which the condition holds true) after a XOR gateway. To ensure well-formedness, a default sequence flow must be included, which is selected when the conditions of all the other flows do not hold true.

The purpose of the **Data Artefacts** is to show the information and information objects required or produced by each activity or event during the execution of the process. The most common type of Data Artefact is represented by a **Data Object**. Such an object represents a concrete information document, which can be in a physical or electronic form and it is produced or used by an activity or an event during the process instance execution. Such a data object is represented as a file with the top left corner folded, as it can be seen in figure 7. A second type of Data Artefact is a **Data Store**. A Data Store represents a physical or electronic container which contains data objects which need to be maintained once the process instance is finished. Activities and events can read or write to the objects contained inside a data store. A data store is represented using a three-stripes cylinder, the usual symbol of a database [2], and can be seen in the same figure 7.

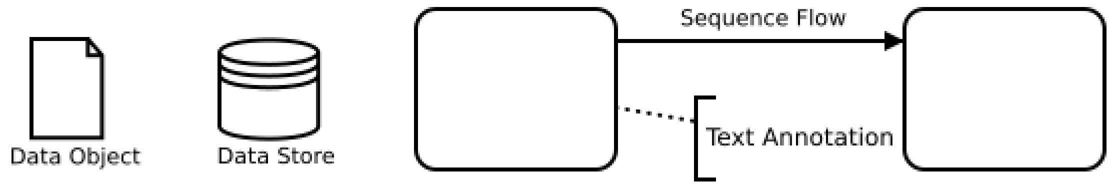


Figure 7: Other BPMN Elements

The objects which can be used to obtain additional information about the process or its elements are called **Artefacts**. The most common type of artefact is called **Text Annotation**, and it is used as an attachment to an activity in order to provide some extra information regarding the way in which the activity is executed or any other aspect related to the activity. An example of text annotation can be seen again in figure 7.

The **Swimlanes** elements are used in order to model the resources which are executing the process activities. There are two types of swimlanes elements: pools and lanes. Although there is no general accepted convention in place regarding the type of resources modeled by a pool or a lane, in general **Pools** are used to represent an entire organization, while **Lanes** are used to divide the organization into specific sub units, such as departments. It is also important to mention that lanes can be used on multiple level, such that a lane can be further divided in other lanes on any number of levels [2].

2.4 Related Work

In this section some similar implementations of BPMN engines will be introduced, contrasting their approach and set of features. All the solutions presented, have as the same primary goal, the ability to animate a BPMN process execution, using the provided engine or software. However the complexity of each such implementation is different, ranging from a simple JavaScript library to that of a commercial tool available on the market.

2.4.1 E2ebridge

The first implementation which can be considered somehow similar to the one presented in this paper is called ***"e2ebridge"*** [13], and consists of a BPMN execution engine implemented in JavaScript, using Node.js. The engine works with BPMN 2.0 models for describing the process, and each such model must be accompanied by a JS file which contains the event handlers for all the BPMN elements included in the process. The engine presents a series of predefined handlers which can be used in order to define the behaviour of the process BPMN elements.

The engine presents some interesting features, such as real-time handling for time events, and full support for inter-pool communication, using message flows. In addition, it provides a logging mechanism with several logging levels, which allow the user to customize the logging process according to his needs. *E2ebridge* also allows for a process to be defined and retrieved later, using the Process Managers used by the engine. In general, each process is identified by its ID, therefore it is important to assign a unique ID to each process. Persistency is assured, allowing the user to store the processes either in the file system, or using MongoDB. The engine supports a large list of BPMN elements, such as: start events, end events, AND gateways, XOR gateways, a large variety of task types, both catch and throw intermediary events, sub-processes, or boundary events [13].

The engine also has some limitations, which are worth mentioning. First, all the start events and end events are mapped in the same way and are equivalent to the normal start and end events. The user has to define the behaviour of such events in the handlers assigned to them in case he intends to use a specialized version of them. In addition, *e2ebridge* is missing two rather important BPMN gateways, which are not supported by the engine, namely: the OR gateway and the event-based gateway, providing support only for the AND and XOR gateways. Another important issue is the lack of support for user interaction with the process, during the execution, due to the fact that the engine does not provide any forms or a similar mechanism which would allow the user to assign or modify values of the business data. The engine is meant to be used in a non-distributed environment and does not provide any features which would allow multiple users to interact with the same process instance.

2.4.2 Camunda-bpmn

The ***Camuda's bpmn-js*** [14] project consists of a JavaScript framework which can be used for parsing, rendering, and executing BPMN 2.0 processes. The framework is composed from three components, each dealing with a different part of the overall functionality. The first such component is the Transformer, which is responsible with the parsing of the XML file containing the BPMN process. After the file is read the information is transformed into a JavaScript object which is used by the other two components. The second component is represented by the Renderer, which is used for the visual representation of the process, being able to render the elements of the BPMN process on an HTML5 canvas. The last functional component of the framework is the engine, which deals with the execution of the process.

The main advantage of the framework is represented by the wide range of functionalities, described earlier, which are provided within the same framework. However, one major drawback is represented by the reduced number of BPMN elements supported by the framework. Camunda does not distinguish between different specializations of the same event type. Therefore, for example, all the start events (message start event, timer start event, etc.) are treated as normal start events. In addition to this, the framework does not provide support for event gateways or OR gateways, the only supported gateways being the AND and XOR semantics. Camunda does not provide any mechanism for the user to interact with the process data during its execution, such as forms to assign values to the data. As in the case of e2ebridge, no support for distributed environments is provided.

2.4.3 Omni-Workflow

One of the most complete tools available for BPMN process execution is called ***Omni-Workflow*** [15], and it provides a complex solution for modelling, executing, and deploying processes. Due to the fact that this paper deals mainly only with BPMN process animation, the other functional parts of the tool will not be described in this section.

Omni-Workflow is a commercial tool that provides a large coverage of the BPMN elements, and which allows the user to interact with the business data available in the process. The tool supports data elements, forms which can be defined for user tasks, scripts for including logic in the service tasks, conditional flow logic, access rules, and notification rules, therefore providing a complete interaction and manipulation of the process during its execution.

However, *Omni-Workflow* does not provide the possibility to deploy the processes in a distributed environment, providing only the option to duplicate an existing process. Therefore, we can say that although from a functional point of view, the tool provides complete support for the BPMN notation, and provides a user interaction with the process mechanism, it lacks the option to allow multiple users to interact with a single process at the same time.

2.4.4 Prime Process Model Animation

The final similar implementation presented in this section is called ***Prime Process Model Animation*** [16] and it is developed at VU University Amsterdam. The system represents a lightweight, self-contained process animation tool, which can be accessed in a browser window, without the necessity to install it or configure it in any way, and without the need of user identification.
Prime provides a friendly and easy-to-use user interface, and multiple animation modes, such as continuous or step-by-step animations. When using the continuous mode, the execution of the process will advance in a continuous manner, stopping only at decision points, where the user has to decide how the animation should proceed. On the other hand, when using step-by-step animation, the user has to execute one element at a time. The tool also provides an animation speed control option, for adjusting the speed of the simulation to the user's needs, along with a tutorial and a set of predefined sample processes.

Prime has a few limitations as well, such as the rather small range of supported BPMN elements: various types of tasks and events, AND gateways, XOR gateways, pools and lanes, or data objects. However, the tool does not provide support for some of the core elements of the BPMN notation, like: OR gateways, boundary events, or sub-processes. In addition, no business data can be attached to the animated process, such that the application does not allow the user to manipulate business data, while executing a process instance. As the previous mentioned implementations, Prime is meant to be used only in single-user environment, and it doesn't provide any support for multiple users interaction.

2.5 Requirements

The main idea of the solution presented in this paper is to create a lightweight, self-contained BPMN process animation engine which would be able to work in a distributed environment, allowing the user to upload a BPMN process and to animate its execution, while being able to define and modify the business data attached to the it. Therefore, the requirements can be divided in two categories: non-functional requirements, represented by the lightweight and self-contained properties, and functional requirements, represented by the abilities to run BPMN simulations, to define the business data through a set of forms attached to the process, and to perform the animation in a distributed environment.

The notion of "lightweight", refers to the fact that the simulator can be used without the necessity to be installed or configured in any way. Unlike most similar solutions, which require some sort of installation or configuration, our tool can be accessed directly in a browser window, and, once the process is loaded into the simulator, the animation can be started right away, without requiring any additional steps. This lightweight property of the simulator provides several advantages. First of all, it assures that the application is cross-platform, due to the fact that it can be run on any operation system which supports a browser.

The second important benefit is represented by the fact that the simulator can be used by persons without a technical background, who might find it hard to configure or install such an application.

The application also has to be self-contained, meaning that it does not require any other piece of software in order to be functional. In other words, the simulator does not have any external dependencies, and it does not interact with other applications, which might cause unexpected problems during its execution. These two properties combined assure that the system is easy to use, providing an accessible solution for animating BPMN process executions.

From a functional point of view, the system combines three important features, which although can be found individually in other similar simulators, put together can provide an innovative way to analyze and animate business processes. The first, and most important, functional requirement is to assure the correct behaviour of the BPMN elements involved in the process modeling. However, it is important to mention that 100% coverage of the BPMN elements is not within the scope of the tool. Instead the most important and frequently used elements are supported, leaving aside more complex elements, such as: message flows, infrequent event types, data objects, etc. The range of elements, which are supported by the tool should allow the user to implement most of the processes required in a company.

The second functional requirement refers to the option of defining the business data attached to the process, and to provide a mechanism for accessing and modifying this data. The mechanism for defining the business data and for modifying its values is implemented using forms, which can be attached to BPMN elements, such that depending on the type of the element, data values can be modified, scripts can be executed, or conditions defined. At first a global data schema is defined, containing all the properties which are to be included in the business data. Once the schema is defined, the user can decide which properties are modifiable when each element is executed, or to attach some conditions to specific elements which can be used during the animation of the process to determine the path to be chosen after a decision point.

The final functional requirement of the system is to provide a mechanism for multiple users to interact with the same instance of the process while being executed. The modifications have to be seen in real time, such that the state of the process is the same for all the users involved in the animation. In addition, each user has to be able to execute the elements of the process during its animation. In other words, the application has to assure that the state of the process instance is synchronized for all the users.

2.6 Discussion

One of the best ways to analyse a business process is through visual animations, with BPMN being the most used notation for modelling such processes. In addition, such animations prove also very useful when redesigning an existing process, in order to make sure that no errors are introduced in the model along with its improvements.

Due to the fact that the notation of BPMN is notably rich in symbols, most animation tools, do not provide support for all the elements, but rather only for a core set which can be used for defining most of the processes implemented within a company. For the implementation of our tool, we also decided to focus on this core set of symbols, leaving aside the elements which do not provide vital functionality when modelling a process, and which appear with a low frequency in process models. The focus has been placed on flow elements, such as events, activities, and gateways, while elements such as data objects, or message flows have not been included in the scope of the simulator.

Although there are many BPMN execution engines available on the market, with different degrees of complexity, ranging from simple frameworks and libraries, to complex commercial tools which can be used during the entire BPM lifecycle, providing more functionality than just animating process models, we believe that none of them provides the exact same properties as our tool. The application presented in this paper is lightweight and self-contained, combining the functionality of a BPMN process execution tool, with the ability to manipulate business data through HTML forms attached to elements, and to perform an animation of a process instance involving multiple users at the same time. In the end, it is worth mentioning that we have the strong confidence that our tool has the potential to be innovative in the field of BPMN simulators.

3 Approach and Implementation

In the contents of this chapter, the main theoretical principles and concepts used for developing the tool will be introduced, along with a detailed technical description of the different modules of the system. The most important algorithms used will be explained, and the entire architecture of the tool will presented, giving the reader a clear image of the different parts which compose the system and their interactions. When necessary, the decisions and choices made during the development process will be presented and motivated, giving relevant explanations as to why one option was prefered over the others. In the end of the chapter, the differences between the single-user, centralized version of the tool and the multi-user, distributed version will be explained.

3.1 Lifecycle of the Animation

As already stated in the previous chapter, the main functionality of the system is to perform animations of a BPMN process in order to allow the user to discover any potential flows in the designed model. However, such an animation can be divided in two different steps, which are executed separately, as can be seen in figure 8. The first step in the lifecycle of the animation is the editing phase, when the business data of the process is defined as a set of different properties which can be attached to BPMN elements, producing and enriched BPMN model. As can be seen in the figure, the inputs of the editing phase are an initial BPMN model and a model schema, which represents the business data. The second phase is the animation phase, when one element is executed at a time. This animation can be executed both in a centralized or in a distributed environment as can be seen in the same figure 8. In the next paragraphs, each of these phases will be introduced to the reader.

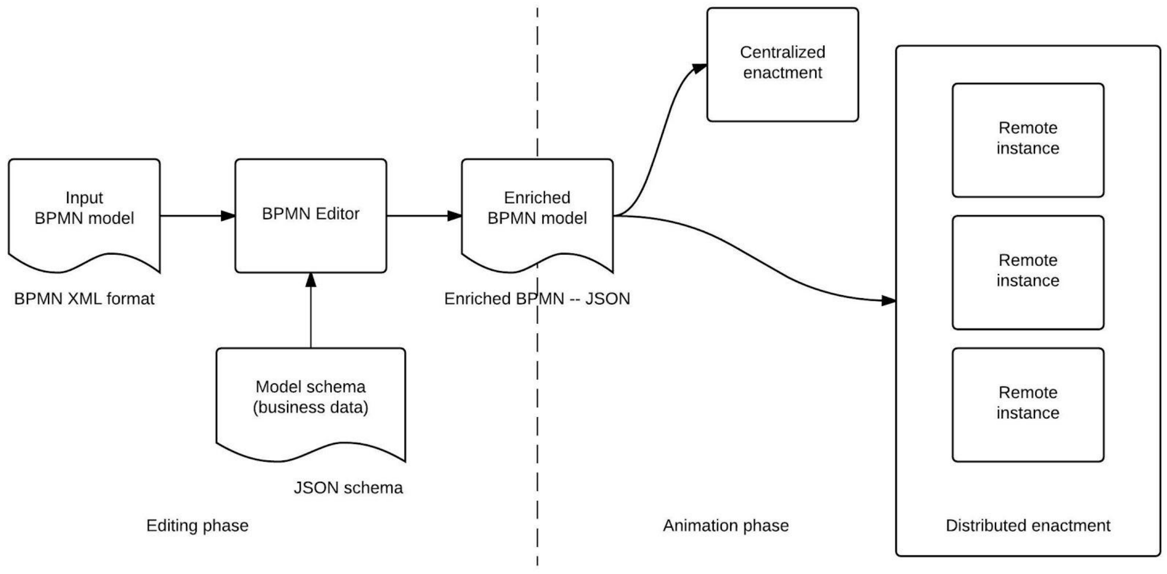
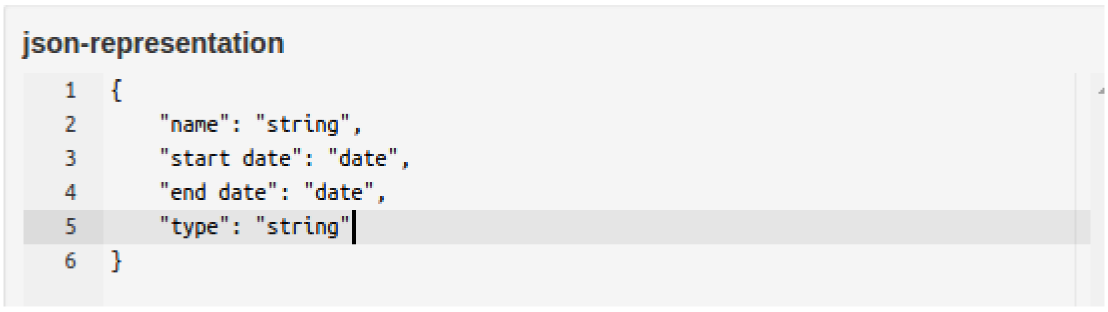


Figure 8: The two photos of the simulation

The first phase of the simulation's lifecycle is represented by the **editing phase**. This phase starts right after the process model is uploaded into the tool, and it lasts until the actual animation is started by the user. During this phase, users can define the global schema which will be used for representing the business data. The global schema is defined as JSON schema [17]. In its basic form, a global schema is a set of variable names and data types, which together represent the entire business data. Figure 9 presents a small sample of a global schema, to illustrate the idea.



```
{
    "name": "string",
    "start date": "date",
    "end date": "date",
    "type": "string"
}
```

Figure 9: An example of the business data defined as a JSON schema

The purpose of this global business data is to contain all the properties which are relevant and which will be used during the execution of the process in the form of data definitions. Once such a schema is defined, users can move to create data definitions on different BPMN elements, such as user tasks, sequence flows, script tasks, or catching message events. For each type of element, a different type of data definition can be attached. Such data definitions are defined before the animation is started, and are based on the business schema's properties. Their purpose varies depending on their type, and they can be used to edit business data or to control the flow of the process during the animation.

The first type of data definition, which can be defined is attached to user tasks, and it offers the users the opportunity to select which properties from the business data are editable when this particular task is executed. The main idea behind this is to be able to provide the user responsible for executing a task with access to modify and visualize only the data to which he is supposed to have access to, while being able to maintain the confidence of the rest of the business information. This is particularly useful when different users assume different roles in the execution of the process, and the process business data contains some sensitive information, such as financial data, which should be available only to the financial department, for example. However, a user from a different department should still be able to access and modify the rest of the business data, while the sensitive data is being concealed from his sight.

A rather similar type of data definition can be defined on catching message events. As in the previous case, on each such event the properties of the business data which can be visualized and modify upon its receiving can be defined before the simulation. Although the data is defined in a similar way, the purpose in this case is rather different, as it represents the data received via message exchange and that needs to be entered into the process.

Another way in which the business data can be modified is through the execution of scripts, without the necessity for the user to set the data manually. Such scripts can be attached to script tasks in order to be executed automatically along with the task. Scripts are important when a property of the business data should be modified automatically by the system upon the execution of an activity without the intervention of the user. It is important to mention that these scripts have to be defined based on the business data properties in order to be considered valid.

Finally, process data is used for determining the flow execution in decision points. Data is evaluated according to conditions attached to sequence flows outgoing XOR-split gateways.

During the animation, after a XOR-split, in the absence of any defined conditions, the user has to select manually the successor element to be executed. However, if some conditions are defined on the sequence flows which connect the gateway to its successor elements, they are evaluated and the next element will be selected automatically by the simulator, if the condition on its sequence flow can be evaluated to a true value. As in the case of scripts, such conditions have to be defined based on the previously defined business data in order for the simulator to be able to evaluate them. In other words, any property used when defining a condition has to be part of the global business data.

Once all the data definitions are defined, the editing phase can be ended and the animation can be started, by pressing the "start simulation" button. During the animation, neither the business data, nor the data definitions can be modified and after the animation is started, the users cannot go back to the editing phase, in order to edit these. One can observe that in order to perform an animation there are two different input elements needed. First, the process diagram in the form of an XML file, and the definitions of business data and element data definitions. It is important to mention that two animations performed using the same BPMN process, but distinct business data and element data definitions, can be seen as two completely different process executions, performed with different input data.

The second phase is the actual **animation phase** when the users can follow how the business process execution would evolve in the given conditions. The animation is started once the "Start Simulation" button is pressed by one of the users. The animation starts at the same time for all the users involved in the process, when the button is pressed by one of them.

The users can progress through the animation by clicking anywhere inside the process diagram displayed in the browser window. With each such action, one element is executed at a time, leaving the process in a new state. In some situations, the user has to make a specific selection between multiple elements, such as selecting the next element to be executed after a XOR gateway, when the data model is underspecified, and does not provide any conditions for automated path selection. During the animation, when the executed elements, have some data definitions attached, where the user can modify manually data values, some forms are displayed in order to allow the user to modify the desired values. The animation can continue only after the window form has been closed. In addition, all intermediary events have to be specifically selected by the user, in order to allow him to simulate different scenarios, as to when such an event may occur. The animation ends when one of the end events is executed and there are no other process elements enabled for execution. In the case when there are multiple elements enabled for execution at the same time, the elements which will be executed during the next step is chosen on LIFO principle.

The two phases described in this section can be seen as the lifecycle of the animation, due to the fact that after the animation phase is over, a new diagram can be loaded in order to start again with the editing phase. The animation phase, can be strongly influenced by the data defined in the previous phase, therefore we can say that the two phases are not independent from each other, but rather that together they form the animation cycle of the system.

3.2 Semantics of BPMN

This section focus in describing the properties of individual BPMN elements and their behaviour. The elements which are supported by the tool will be presented, and, when necessary, their behaviour will be explained and implementation decisions will be motivated.

3.2.1 Supported BPMN Elements

As already stated in the previous chapter, the simulator does not provide a 100% coverage of the BPMN 2.0 elements. This is, in part, due to the fact that the BPMN 2.0 standard contains over 100 distinct symbols which can be used when modelling a business process. On the other hand, during the implementation of the simulator, the decision was taken to focus on implementing and enriching the animation experience and functionality rather than supporting a large amount of elements. However, from each important BPMN element category the elements which tend to appear in process design with a higher frequency, and which are generally considered part of the core set, were selected for implementation. In the remaining of the section all the supported BPMN elements will be presented, along with some justifications as to why these particular elements were selected over the others.

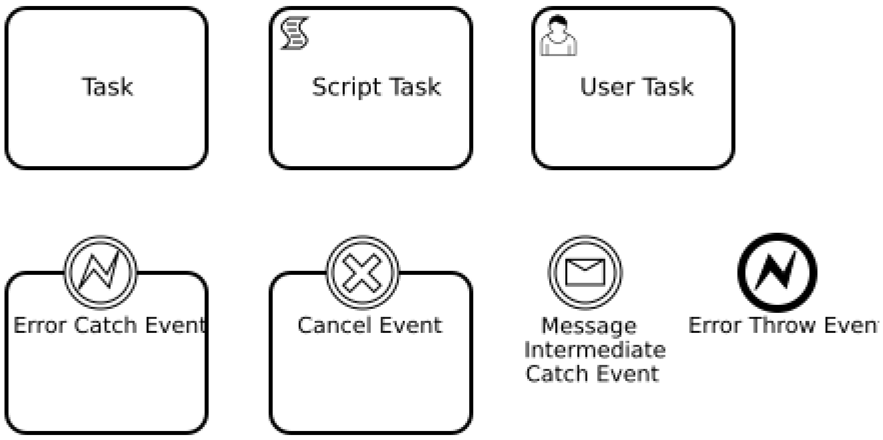


Figure 10: Supported activities and events

The first category of elements which are supported by the simulator is represented by **Activities**. In this category we can identify two separate sub-categories which can be used when simulating a process using the tool: tasks and sub-processes. The simulator supports three types of tasks: regular tasks, script tasks, and user tasks. The regular **Task** (figure 10) was chosen for implementation due to the fact that it represents the most common activity used when designing BPMN business processes. It has no special properties and it represents the most basic type of activity executed in a process. The other two types of tasks chosen to be supported by the simulator, were chosen due to their special functionality and significance. The **Script Task** (figure 10) was implemented in the tool due to its ability to execute scripts while being executed. Such scripts are useful during the simulation in order to modify the content of the business data. The second type of specialized task is represented by the **User Task** (figure 10). This BPMN elements is supported, due to its nature of usually being executed by user with the help of an automated system. This property has made the User Task the ideal element for presenting the user with the opportunity to manually modify the values of different properties of the business data, when executing such an element. All the other specialized types of tasks are treated as simple tasks by the simulator during the execution of the instance, which means that no special properties are taken into account, but the simulation is able to cope with the presence of such elements in the BPMN process model.

In addition to tasks, the simulator also supports sub-processes. IN BPMN process modelling we can identify two types of sub-process elements, namely expanded sub-process and collapsed sub-process. The main difference is that the expanded sub-process represents a sub-process which is included in the same process diagram, while the collapsed sub-process element is usually connected to a different diagram representing the sub-process. Due to simplicity reasons, the simulator only supports the **Expanded Sub-process** element, such that the entire sub-process is included in the same diagram as the parent process. One example of an expanded sub-process can be seen in figure 11.

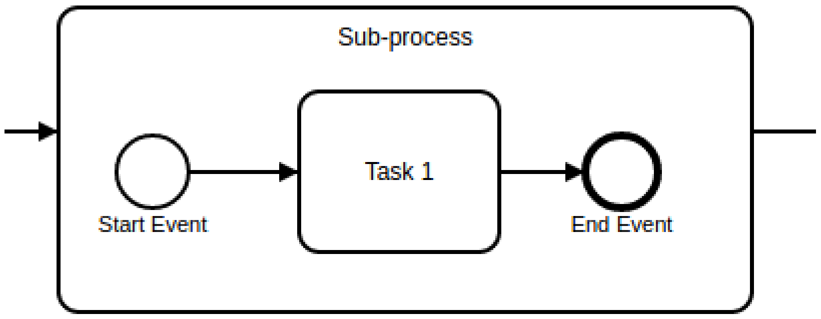


Figure 11: Example of an expanded sub-process

The simulator supports multiple levels of nesting for sub-processes, such that a sub-process can contain other sub-processes within itself. The sub-process can contain any type of elements which are supported by the simulator, with the only restrictions that it must start with a start event, and have as the last element an end event (or more end events, placed on different paths). The sub-process can have some special types of boundary events attached to it which will be presented later in this section.

The second important category of BPMN elements is represented by the **Events**. In BPMN 2.0, there are many different types of events which can be used when modelling a process, and they can be divided both, by their role in the process (start, intermediate, or end), and by their type (message, time, etc.). In addition, some types of events, like the message event can be further divided in catching message events, when a message is received during the process, and throw message events, when a message is sent during the process towards another entity. In our system, most messages are treated the same way, in the sense that they have to be selected by the user in order to symbolize their arrival or their departure. However, some types of messages, have some different properties during the simulation, which are meant to better represent their meaning. One important message during the simulation is represented by the **Intermediate Message Catch Event** (figure 10). This type of event allows the user to modify or assign values to some business data properties, in order to symbolize the arrival of the data attach to such a message which is to be used during the process execution. The same behaviour is implemented for the **Message Start Event**, such that data can be received upon the start of the process. The other types of start, intermediate, and end events are treated in the same way as simple events by the tool. It is also worth mentioning that a process may end with a simple end event, with a terminate end event or with another type of end event.

A special category of events is represented by the **Boundary Events**, which can be attached to different types of activities, such as tasks and sub-processes. The simulator treats most types of boundaries events the same, with a few exceptions, which present a special behaviour. Such special boundary events which are treated in a different way by the simulator are the **Cancel Event** (figure 10) and the **Error Events**. The cancel event is usually attached to a sub-process and it is used to cancel the execution of the entire sub-process when selected.

Another type of special boundary event is represented by the **Catching Error Event** (figure 10). Unlike most events involved in the simulation, this event cannot be selected by the user, and it is only activated when a **Throw Error Event** (figure 10) have been executed inside the sub-process.

The last important category of BPMN elements, supported by the system is represented by the **Gateways**. Our simulator supports the most frequently used gateway elements in business processes design, using the BPMN 2.0 standard. These are represented by the **AND Gateway**, **XOR Gateway**, **OR Gateway**, and the **Event-based Gateway**. The AND, XOR, and OR gateways can be either a split gateway or a join gateway, each having different purposes during the simulation. On the other hand, the event-based gateway can be only a split gateway, and usually an XOR join gateway is used to re-merge the execution path. One important limitation which the current tool has is that a gateway cannot be at the same time both a split and an or gateway. In order to avoid this situation, two consecutive elements have to be used, one join gateway followed by a split gateway.

In addition to the flow objects presented until now, the tool also supports some additional elements which are vital in order to be able to run an animation. The most common of these elements, is represented by the **Sequence Flows**, which are used in order to connect different flow elements, with the purpose of creating successor-predecessor relationships. The tool is also supporting swimlane elements, such as **Pools** and **Lanes**, with the mention that, such elements are treated simply as containers for the other elements. At the moment, the tool can only run processes which contain one pool, therefore more advanced functionalities related to such elements, such as collaborations between different process pools are not supported at this time by the system. The reason while pools and lanes have been introduced in the simulator as elements, is to allow the user to upload for simulation processes which were modelled using such elements for representing entities such as companies, or departments.

Up to this point, the elements supported by the tool have been presented and their choice have been motivated to the reader. One can easily observe that there are many BPMN elements which are not supported by the system, therefore they were not included in the above descriptions. The reasons for which various BPMN symbols have not been included in the list of supported elements are diverse and usually different for each individual element or category of elements. In the last part of this section, some reasons will be presented in order to justify the absence of some of these not included BPMN symbols.

The first types of elements which are not supported by the tool are represented by some specialized types of activities, such as multi-instance task or standard loop task. The reason why these elements were not implemented in the simulator is in part, due to the fact that they are not present with high frequency when modelling business processes, and in part to the fact that their behaviour is not a trivial one to implement. Moreover, as already stated in the previous paragraph, the simulator lacks the concept of inter-pool collaborations, which would allow the user to define multiple entities, such as companies, which would interact when executing a process. The collaboration concept has not been seen as a high priority for the simulator, due to low relevance for the scope and objectives of the tool. Due to this reason, all the afferent elements of such collaborations, such as message flows are not supported by the tool. The last major category of elements which are not included in the simulator is represented by the data artefacts. As in the previous case, these were not included in the tool because they do not offer any enrichment for the requirements and purpose of the application.

3.2.2 Element's Lifecycle

During the animation, there are multiple states in which any flow object of the model can be found in. Depending on the case, each state can reflect the actions which the user has already executed upon the element, or the actions which can be executed by the user at the current point in the animation. Each such state is visually represented by colouring the BPMN element in a different colour, and have a different logical meaning in the animation's context. At any given point in time, each flow object element of the process has to be in exactly one state. These states are only applied to the flow objects, namely: tasks, events, and gateways, while the other types of BPMN objects are stateless during the animation. In other words, all other types of elements do not have the concept of state defined and do not change their properties during the animation. A schema with all the existing states of the elements during the animation, along with the transitions between them can be seen in figure 12. In the rest of this section each individual state will be presented in details, along with a visual example of its representation during the animation.

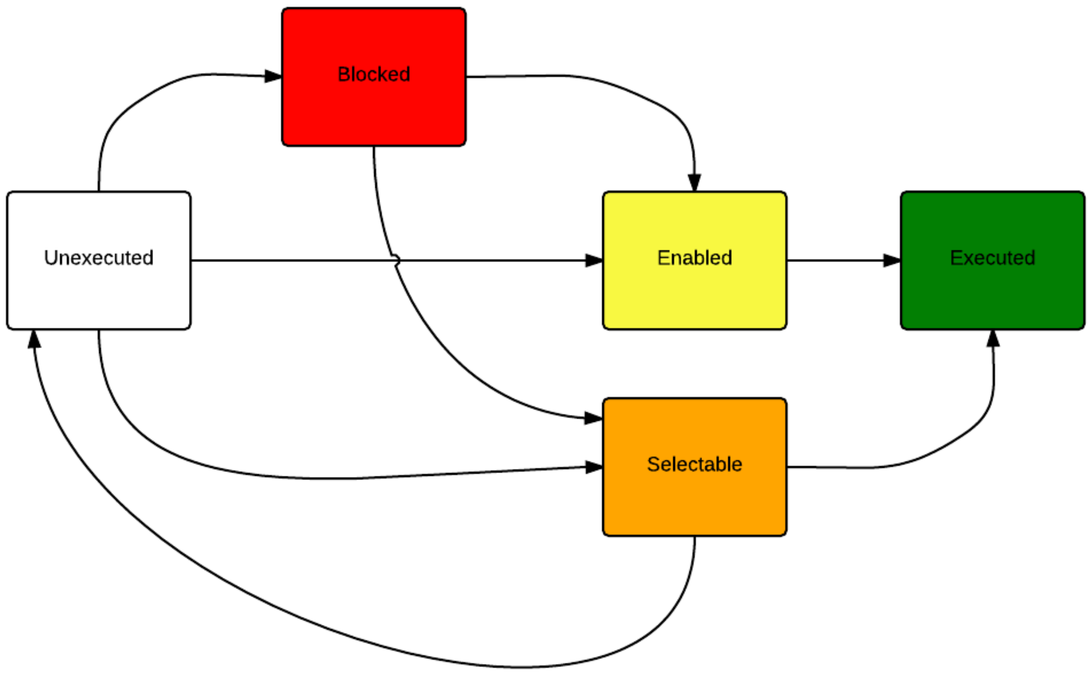
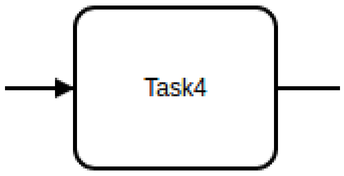
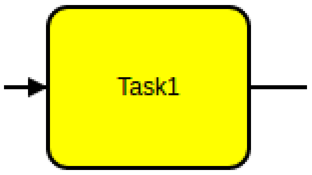


Figure 12: Element State Transitions

The **Unexecuted** state is the state in which an element is before any type of action have been performed upon it. In the beginning of the animation, all elements are unexecuted, therefore we can say that this is the initial state of an element. Intuitively, when an element is unexecuted, we can say that the animation has not yet reached this element, or that the element is on a path which will never be executed during the current process execution. The only state from which an element can go back to unexecuted is the selectable state, in case the user does not select the element for execution. From all the other states, an element cannot go back into unexecuted. Figure 13.a presents a task which is in the unexecuted state.





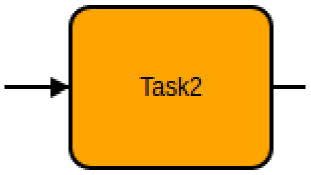


Figure 13.a: Unexecuted Task   Figure 13.b: Enabled Task   Figure 13.c: Selectable Task

The **Enabled** state symbolizes that the element is ready to be executed. Once an element is brought to this state, sooner or later the element will be executed during the animation, therefore the only state in which an element can move from enabled is the executed state. During the animation, multiple elements can be enabled for execution at the same time. However only one enabled element is executed during each step of the animation. Usually, an element is brought into the enabled state, once its predecessor or, depending on the element, predecessors have been executed, such that there is no impediment for the execution of the current element according to BPMN principles. In the beginning of the animation, usually the start events are the only elements which are enabled. The animation is considered to be finished, when no elements are enabled or selectable, therefore the animation cannot be brought into a new state. In figure 13.b an enabled task can be seen.

Not all elements move from the unexecuted state into enabled state. Some elements require a specific selection action from the user in order to be executed. We say that these elements are in the **Selectable** state. There are multiple situations when an element can be found in this state, and usually it depends upon the type of element. The most common situation is after and XOR split gateway is executed. The successors of the gateway are put into selectable state, such that the user can select which path should be enabled from the possible alternatives. In this scenario, when one of the selectable successors is selected, the others are set as unexecuted, and the animation will proceed with the selected path. Another quite similar situation when successor elements are marked as selectable is after an OR split gateway is executed. However, in this situation, the user will be presented with an option to execute or not to execute for each of the successors, regardless if any successor was previously executed or not. During the animation, whenever an intermediate message element is reached, this element is marked as selectable such that the user can decide at which point this element will be executed. However, in this case, the element cannot be brought back into an unexecuted state, due to the fact that sooner or later, the event is bound to happen. One special case of events activation is when an event-based gateway is executed. The successors of this gateway have to be only events, from which, only one will be selected by the user. In this case, the events that follow will be marked as selectable, until the user selects one of them as the continuation path. Another special situation, when the elements are in the selectable state is when a task has a boundary event attached to it. In this case, when the task is reached during the animation, both the task and the event are marked as selectable, rather than the task being enabled, such that the user can select which path to follow. The last important situation when elements can become enabled is when the execution path enters a sub-process which has boundary events attached. During all the time that the execution path is inside the sub-process, such events will be selectable. Depending on the situation, an element can move from a selectable state into an executed state, or into an unexecuted state. Figure 13.c shows a selectable task during the animation.

One very uncommon state during the animation is represented by the **Blocked** state, due to the fact that this state is possible in only two particular situations. The meaning of this state is that an element has been reached during the animation, but the conditions for its enablement are not yet met. In other words, some predecessors of the element still have to be executed before the element can become enabled or selectable. There are only two BPMN elements which can be found in a blocked state: OR join gateway and AND join gateway. These elements are put into a blocked state, when one of the paths which they synchronize is executed all the way until the gateway, but some other similar paths have not yet been executed. In this case, the gateway cannot be executed, due to the fact that it has to wait for the other branches to be executed as well. However, we cannot keep the gateway in the current unexecuted state, because we have to signal to the user, that the other branches have to be executed, before the gateway can be enabled. From the blocked state, elements usually move into the enabled state, or in some rare situation into the selectable state. Figure 14.a presents a blocked OR gateway.

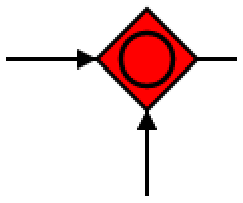


Figure 14.a: Blocked OR Gateway

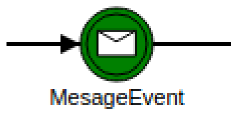


Figure 14.b: Executed Message Event

The last state in which an element can be is the **Executed** state. If the process model does not contain loops, which will bring the execution flow back to an already executed element, this is the final state of an element. If case, the process contains such loops, then the element can be re-enabled once executed. The meaning of this state is that the element has been executed, that its side effects upon the process have already taken place, and that its successors may be enabled for execution. At the end of an animation, all the elements of the process have to be either in an executed state or in an unexecuted state, in case they are on a branch which was never executed during the animation. In figure 14.b, an executed message event can be seen.

The states of the elements represent a very important aspect in the overall execution of the animation, due to the fact that the set of the states of all elements at a given point in time represent the overall state of the process. The state of the element is also responsible for its behaviour, due to the fact that the same element may have different behaviours during the simulation, depending on the state in which it is.

3.2.3 Behaviour of the Elements

In this sub-section the behaviour of the various elements supported by the simulator will be presented from the process execution point of view. This section does not deal with the element data definitions and the forms attached to each element, as these will be presented in detail in a later section. The purpose is to explain the behaviour of each element during the animation and to provide an insight into the internal representation used for the process execution animation.

Before starting to explain the individual behaviour of each element, it is important to introduce the concept of token, which is used by the simulator for evaluating the state of each element. In the BPMN context, the token represents an object which is placed on sequence flows, in order to represent the fact that the source element of the sequence flow has been executed. Most BPMN elements consume tokens from their incoming connections and produce tokens on their outgoing connections. Our implementation is based on the concept of tokens, and a list of all the sequence flows containing tokens is permanently maintained and updated by the application during the animation.

Most of the BPMN elements supported by our tool, have a really straight forwards behaviour, from the token-flow point of view, in the sense that they consume a token placed on their incoming sequence flow(s) and they produce a token on their outgoing flow(s). Such a behaviour can be observed on all types of **task** elements and **intermediate events**, during the animation. There are two special types of elements which behave slightly different from the previously mentioned ones, namely the start events and the end events. The **start events**, do not consume any token, due to the fact that they do not have any incoming connections, while the **end events** do not produce tokens, since they do not have any outgoing connections.

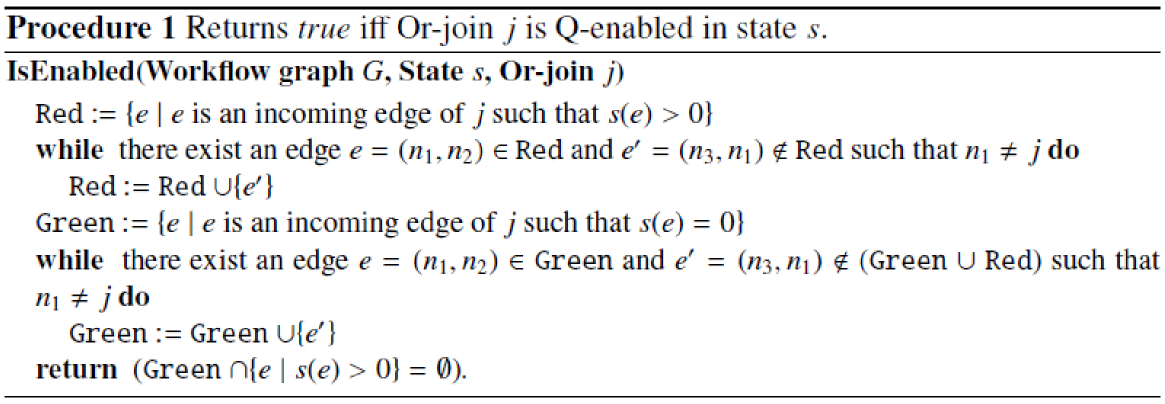
From a token-flow perspective the most complex BPMN elements are the gateways. Each type of gateway has a different behaviour in terms of consuming and producing tokens. The **XOR split gateway** consumes a token from its incoming sequence flow and initially produce one token on each of the outgoing connections. However, when the user selects one of the successors to be executed, the others are set back into an unexecuted state and all the tokens are removed from their incoming connections in order to leave the process in a consistent state. The **event-based gateway** operates in a similar manner from a token-flow perspective. On the other hand, a **XOR join gateway** is enabled once a token is placed on one of its incoming connections. After the gateway is executed, a token is produced on its outgoing sequence flow. If tokens are placed on multiple incoming connections, the process presents a lack of synchronization problem, and the gateway will be executed multiple times, due to the fact that only one token is consumed upon its execution.

In a similar manner as its XOR counterpart, the **AND split gateway**, consumes a token from its incoming connection, and produces tokens on all the outgoing sequence flows connected to it. The main difference from the XOR split gateway is that, since all the elements that follow the AND split gateway will be executed, sooner or later during the animation, the tokens are not removed from the outgoing connections of the gateway when one of its successors is executed. Instead, this successor element will only remove the token from its own incoming connection. A more complex behaviour can be observe in the **AND join gateway**. This type of gateway expects a token on every incoming connection in order to be enabled for execution. Therefore, we can identify three different situations when describing the behaviour of this element. While not even one of the incoming sequence flows contain a token, the gateway remains in an unexecuted state. When at least one of the incoming sequence flows of the gateway contains a token, the gateway is evaluated for enablement. However, since the AND join gateway consume tokens from all its incoming connections, when executed, if not all such connections contains tokens, the gateway is put in a blocked state. While in a blocked state, the condition for the gateway enablement is reevaluated at each step of the animation. When finally, all the incoming sequence flows contain a token, the gateway is set as enabled. When executed, it consumes all these tokens and produces a token on its outgoing connection flow.

The last pair of gateways to be discussed is represented by the OR join and split gateways. The **OR split gateway** consumes the token placed on its incoming connection and produce tokens, at first on all of its outgoing connections. For each successor of an OR split gateway, the user has the option to execute the element or to disable it, returning it into an unexecuted state, and removing the token from its incoming connection.

By far, the element with the most complex behaviour is represented by the **OR join gateway**. The theoretical description of the OR join gateways says that this gate must wait for tokens on all its incoming branches were these tokens will eventually arrive [18]. The main challenge in implementing such a behaviour is being able to determine, in a formal way, when the gateway should wait for a specific incoming connection to receive a token, and at what point in the animation it can be considered that such a token will not be produced by the process. It can be noticed straight away that, unlike the previously presented gateways, the OR join gateway has a non-local semantics [18] [19], meaning that in order to determine if it can be executed we need to look further down the process than just on the immediately incoming edges. Due to the ambiguous description of the OR join gateway in the BPMN Specification, there is no formal definition of this gate that is widely accepted by all the parties involved in BPM, like in the case of the other gateways.

Our implementation of the OR join gateway is based on two main sources, which provide two formal definitions for this gateway: [18] and [19]. The article presented in [18] was mainly used as a starting point for the implementation and in order to properly understand the challenges and the theoretical concepts needed for formalizing the OR gateway definition. The actual algorithm used for determining the enablement of the XOR join gateways used by the simulator is defined in [19] and can be seen in figure 15.



Procedure 1 Returns *true* iff Or-join j is Q-enabled in state s.

```
IsEnabled(Workflow graph $G$, State $s$, Or-join $j$)
  Red := $\{e \mid e$ is an incoming edge of $j$ such that $s(e) > 0\}$
  while there exist an edge $e = (n_1, n_2) \in$ Red and $e' = (n_3, n_1) \notin$ Red such that $n_1 \neq j$ do
      Red := Red $\cup \{e'\}$
  Green := $\{e \mid e$ is an incoming edge of $j$ such that $s(e) = 0\}$
  while there exist an edge $e = (n_1, n_2) \in$ Green and $e' = (n_3, n_1) \notin$ (Green $\cup$ Red) such that $n_1 \neq j$ do
      Green := Green $\cup \{e'\}$
  return (Green $\cap \{e \mid s(e) > 0\} = \emptyset$).
```

Figure 15: Algorithm used for enablement of XOR gateways - taken from [19]

As it can be seen from the pseudocode presented in figure 15, the algorithm uses two different sets of edges in order to determine the enablement of the XOR gateway. Furthermore, we can say that the algorithm is composed from two different parts. First, in the set called “Red” are put all the edges which are reachable starting from incoming connections of the gateway which contain tokens, by going backwards in the process. It is important to say that the search is stopped when the gateway itself is reached by going backwards, in order to avoid including the incoming connections of the gateway which do not contain tokens. The second part of the algorithm computes the “Green” set, as the set of edges which are not already included in the “Red” set and which can be reached by going backwards in the process, starting from the incoming connections of the gateway which do not contain tokens.

Once again, the search is stopped upon reaching the gateway itself, such that the incoming connections of the gateway which contain tokens are not included in the set. The enablement of the gateway is represented as the condition that none of the edges included in the “Green” set contain any tokens [19]. The full explanation of the algorithm can be found in the mentioned article [19], along with a complete demonstration of its validity. As in the case of the correspondent AND gateway, the OR join gateway can be found in three different states. If not even one of its incoming connections contain a token the gateway will remain in an unexecuted state. When at least one such connection contains a token, but the condition for the gateway enablement is not true, the gateway will be placed in a blocked state. The meaning of this state is that the gateway is still waiting for some tokens that will eventually arrive. When the enablement condition is evaluated to a true value, the gateway is set as enabled. In this situation all the tokens that could have arrived on its incoming connections have already arrived. Upon its execution, the gateway consumes all the available tokens from its incoming connection and produces a token on its outgoing one.

Another special category of elements is represented by the **boundary events**, which are attached to tasks, due to the fact that these events do not have any incoming connections. However, the process would be left in an inconsistent state if the tokens placed on the parent’s incoming connections would not be removed. Therefore, when a boundary event is executed, the tokens placed on the incoming connections of the parents are removed, while a token is produced on each of the outgoing connections of the event. Such boundary events become selectable at the same time as the parent element, and if the parent element is executed they are put back into an unexecuted state.

A special case of boundary event is represented by the **cancel event** attached to a sub-process. In this case, the cancel event becomes selectable when the start event of the sub-process is activated. In case the sub-process is executed until the end, the cancel event returns to a unexecuted state when the end event of the sub-process is executed. While the execution flow is still inside the sub-process and the cancel event is selected, the enabled, selectable, and blocked elements from the sub-process are returned to an unexecuted state, and the tokens placed on their incoming connections are removed. Upon its execution, the cancel event produces a token on its outgoing connection.

The situation is a bit different when we talk about an error event attached to a sub-process. The **boundary error event** is not activated when the animation reaches the sub-process, like the cancel event. In order for a boundary error event to be enabled it has to be matched with an **error end event** inside the sub-process. In order for two such events to be matched it is required that they have the same name and that they belong to the same sub-process. If the end error event is executed during the simulation, the token is removed from its incoming connection and the boundary error event is marked as enabled. Therefore, upon its execution, the error boundary event will not consume any tokens, but it will produce tokens on its outgoing sequence flows.

When during the animation a **sub-process** is reached, the application removes the token from the incoming sequence flow of the sub-process, and the sub-process start event is enabled. Afterwards, the animation will continue in the normal way executing one element of the sub-process a time. When the sub-process end event is executed, a token is placed on the outgoing connections of the sub-process element. It is worth saying that apart from the entry and exit points into and from a sub-process, the application does not distinguish between an element that has as parent the main process and an element that is placed inside a sub-process, both being executed in the same way.

3.3 Architecture

In this section, the simulator will be explained from a technical point of view, along with the technology choices made during the development. The main differences between the distributed and the centralized versions of the tool will also be presented and some implementation details will be discussed in order to provide a good overview image of the tool.

3.3.1 General Presentation

The simulator is implemented in JavaScript [20], using Node.js [21] for the implementation of the server-side part. The implementation uses bpmn-js [22] library in order to parse and display BPMN process diagrams and the jQuery [23] library for rendering information on the screen and manipulating visual elements. Although the bpmn-js library provides a complex set of functionalities, for our implementation we rely only on: the abilities to parse the XML file containing the diagram and extract the BPMN elements, use of the provided BPMN element structure, and the diagram rendering functionality. When a diagram is loaded into the system, the xml is parsed and the elements are extracted. Next, the simulator creates some lists and maps which are used during the animation, such as the adjacency list or the sequence flows map. In addition, the source element(s) of the process are identified and the initial state is created.

The simulator uses the concept of state in order to describe the status of the process at a given point in time. In order to define and work with states, the Redux [24] library it is used by the system. Redux is a JavaScript library which can be used in order to manage the states of the application and to create a new state based on the current one. The library uses a user defined reductor which takes as parameters the current state and a user-defined action, and based on these, it is able to create the new state of the system. The old states are never overwritten, such that each new state is added on top of the old ones and a history of the states is created. In our case, the state represents the current animation progress of the process, and a new state is created whenever a new element is executed. The state of the simulator will be described in details in section 3.3.3.

3.3.1.1 Structural Architecture

Before starting to talk about the architecture of the application, it is important to mention that two different versions of the simulator were developed: a centralized single-user version and a distributed multi-user version. Although the working principles of the two versions are identical, the distributed version adds a few additional elements to the architecture. While for the single-user version, the entire simulator is structured as a single application which can be accessed in a browser, for the distributed implementation, the entire architecture is split into two different applications: the client and the server. The user connects through the browser to the client application which communicates with the server application. In this case, the communication between the two aplications is implemented using socket.io [25] and socket.io-client [26] JavaScript libraries.

3.3.1.1.1 Centralized Implementation

The structural architecture of the single-user version of the simulator can be seen in figure 16. The entry point of the application is represented by the ***app.js*** script which is responsible for loading the diagram when the user drags and drops it inside the browser window. Once the diagram is parsed, the same script will create a process instance using the class defined in *bpmn-semantics.js* script file and will pass the process data to the process instance. As part of this process initialization, the events attached to the interface buttons are created, the editor objects used by the forms are defined, and the data definitions for the elements are initialized. Apart for this initialization part, the script is also responsible for handling all user events and to handle the Redux states, by defining a reductor for computing the next state of the process. The main purpose of this script is to act as an interface between the user and the other scripts of the application. However, it is not responsible for dealing with the BPMN semantics, and it does not know anything about how the elements should be handled during the process execution.

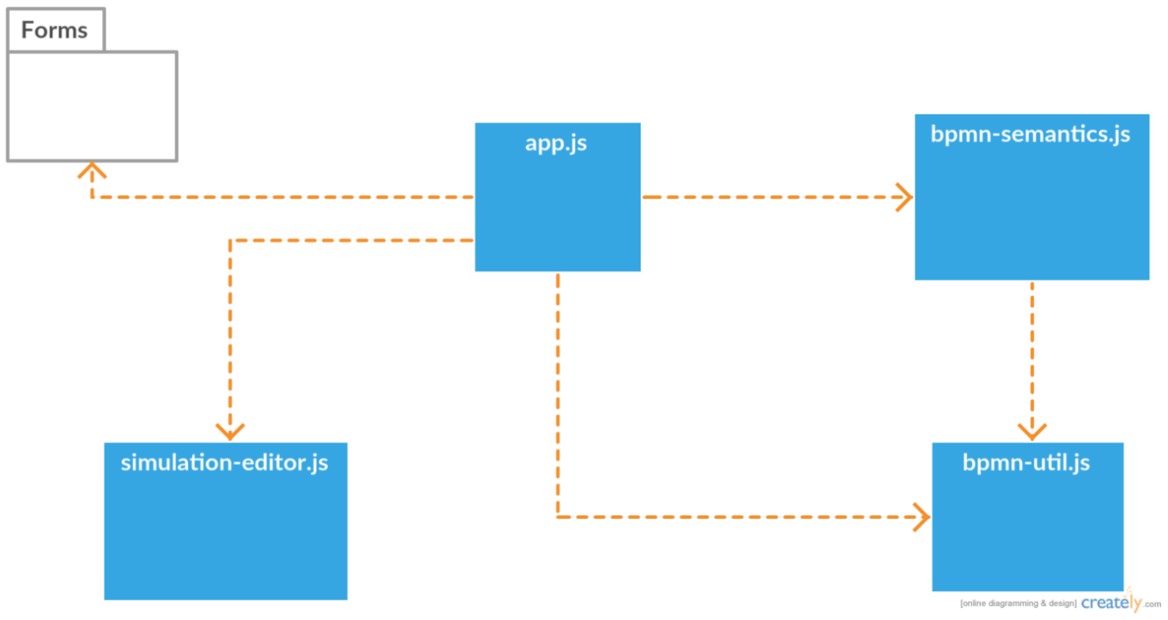


Figure 16: General architecture of the single-user version of the Simulator

The second most important script of the application is represented by the ***bpmn-semantics.js*** script file. In this script file, the ***BPMNProcess*** class is defined, which is responsible for handling everything related to the BPMN semantics, such as individual handling of different elements, or gateway activation conditions. The class represents the process instance from a BPMN point of view and is responsible for creating the next state of the process, based on the element which is being executed at the current step, by determining the enabled, executed, blocked, and selectable elements. The class also encapsulates the functionality for selecting and deselecting an element, and different test functions which can be used in order to test the type of a given element.

The ***bpmn-util.js*** script contains the ***BPMNUtil*** class. This class contains a series of static methods, such: remove element from array, test to see if an array contains a specific element, intersection of two arrays, or test array equality. These functions are of general use, and defined in a separate class for reusability purposes and readability of code.

The last important script of the application is called ***simulation-editor.js***. In this script file can be found the ***SimulationEditor*** class, which encapsulate the functionality required by the forms of the application. Such functionality includes the creation of the JSON schemas for different form editors, based on a set of properties, or determining which are the selected and non-selected properties for a task, based on the input JSON schema. The class instance is instantiated in the main *app.js* script along with the process instance.

The window forms of the application are not included in this section, as they will be presented later in a separate section.

3.3.1.1.2 Distributed Implementation

Although the architecture of the multi-user distributed version is in general terms similar to the centralized one, now the functionality is split between the client and the server applications. In addition, two new concepts appear in this version: the communication mechanism between the two applications, and the synchronization mechanism of all the available clients. One of the main synchronization challenges was the ability to cope with the connection of new clients at any moment and during any phase of the process execution. Such clients would require to be set up to date with all the modifications already done before their connection, such as elements data definitions or the current state of the process animation. The structural architecture of the multi-user version of the simulator can be seen in figure 17.

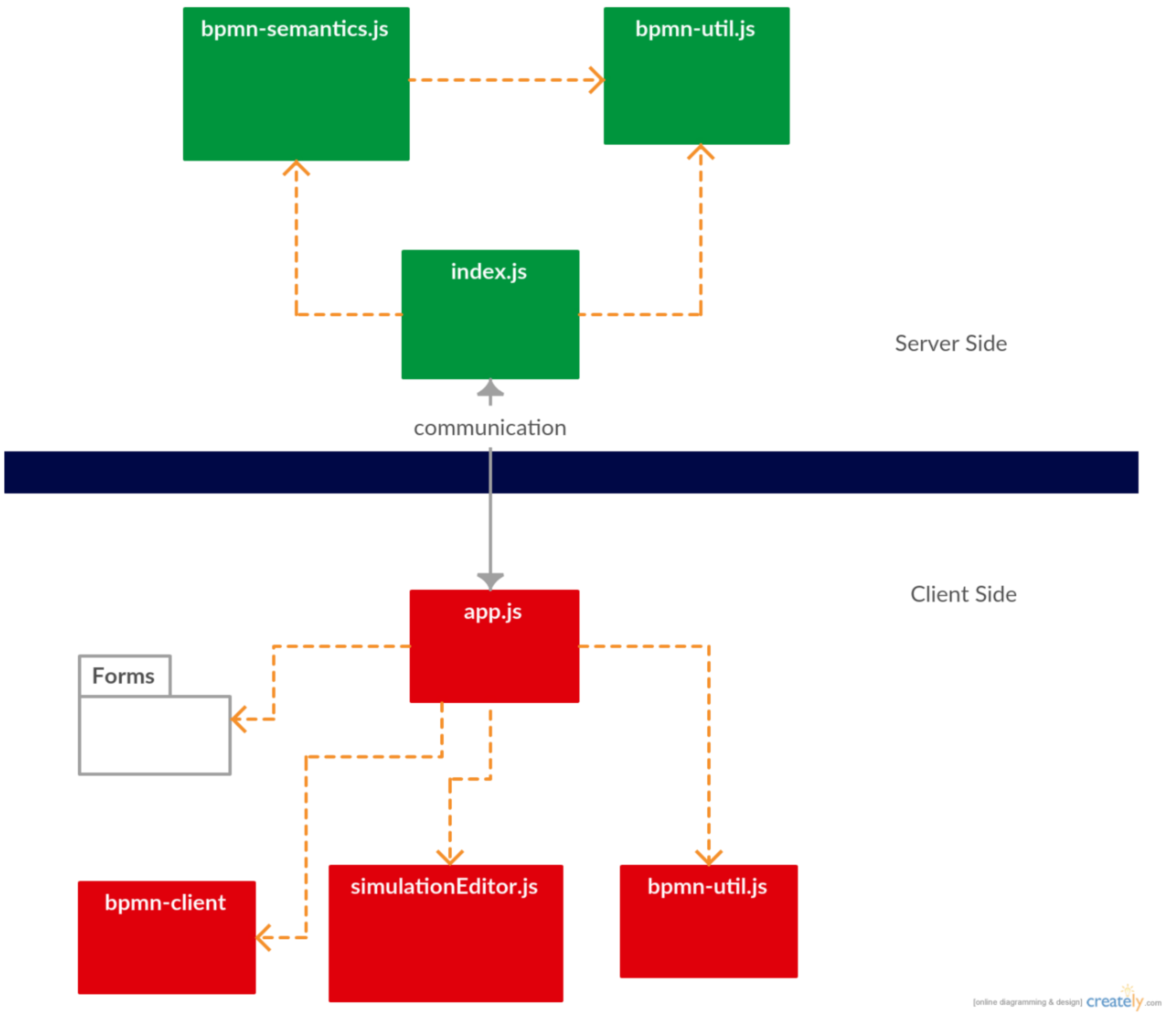


Figure 17: General architecture of the multi-user version of the simulator

In the case of the distributed architecture, the functionality which was previously assigned to the *app.js* script in the single-user version is now split between the two applications: the client side and the server side.

The server side entry point is represented by the ***index.js*** script file. The script is responsible for parsing the diagram received from the client application, to create the process object and to pass the process data to it. In addition, the script also defines the reductor used by Redux in order to manage the states of the process and defines the data definitions for the BPMN elements of the process. One additional functionality, which was not defined in the centralized version is to receive and broadcast messages from and to the clients. As it can be seen in the architecture diagram (figure 17), the *index.js* script is responsible for handling the communication with the client application.

The ***bpmn-util.js*** script which contains the ***BPMNUtil*** class is almost identical to the the one described in the single-user architecture section. Only small modifications have been performed in some functions, which were imposed by the new distributed model. The situation is quite similar when we talk about the ***bpmn-semantics.js*** script, which contains the ***BPMNProcess*** class. The only modifications done were imposed by the need of the new architecture, but the overall functionality and role of the class remains exactly the same as the one described in the previous section.

The client side entry point is represented by the ***app.js*** script file which is responsible for reading the xml diagram file which the user can drag and drop inside the browser window. Once the diagram is read, the xml is sent to the server side which is responsible for parsing it. In addition, the events attached to the user interface buttons are defined and the form editor objects are initialized. The script is also responsible for opening the window forms whenever, the current state of the process execution requires this. In this distributed version, all the forms are handled on the server side and only the result the forms, such as the global business schema is sent to the server side. In the same script is also performed the visual rendering of the process and the handling of the user events received during the process execution. The last task performed inside this script is to assure the communication with the server, by sending information to the server, when a modification is performed and receiving messages from the server with the modifications done by other clients.

The ***simulationEditor.js*** and ***bpmn-util.js*** scripts are almost identical to the versions presented for the centralized implementation. The reason why *bpmn-util.js* script can be found in both, the client application and the server application, is that both applications require the functions defined in this file, due to their generality and to the fact that they do not operate with any application specific behaviour.

The last script of the client side application is the ***bpmn-client.js*** script, which contains the ***BPMNClient*** class. This class encapsulates some of the test methods, used for determining the type of an element, which were previously included in the *bpmn-semantics.js* file in the centralized version. However, such methods are also required on the client side application, and since the *bpmn-semantics.js* script was assigned to the server application, the solution was to create this new class in order to encapsulate the required functionality.

All the forms used for allowing the user to interact with the process execution belong also to the client side application, due to the fact that all the front-end functionality is included in this application. Only the information resulted from the choices made by the user in these forms is sent to the server application to be broadcasted to all the clients.

3.3.1.2 Behavioural Design

After analysing the implementation from a structural point of view, the next step is to present the behaviour of specific functional parts of the simulator from a technical point of view. The behaviour will be presented in the form of sequence diagrams [27] which will be explained step-by-step. The same pieces of functionality will be presented for both: the centralized and distributed versions. The scenarios selected to be presented through sequence diagrams are: the execution of a user task, the execution of a script task, the execution of a XOR gateway, and the selection of a boundary event.

3.3.1.2.1 Centralized Implementation

The first presented scenario is the execution of a user task. This scenario happens when a user task is the first element in the queue of elements which are enabled for execution, and the user executes the next step of the animation. This scenario can be seen in figure 18. By following the execution, we can notice that, first, the user task windows is displayed to the user before any element is executed. Once the user presses the button to save the changes made and proceed with the animation, the task is executed, and the new state is displayed on the screen.

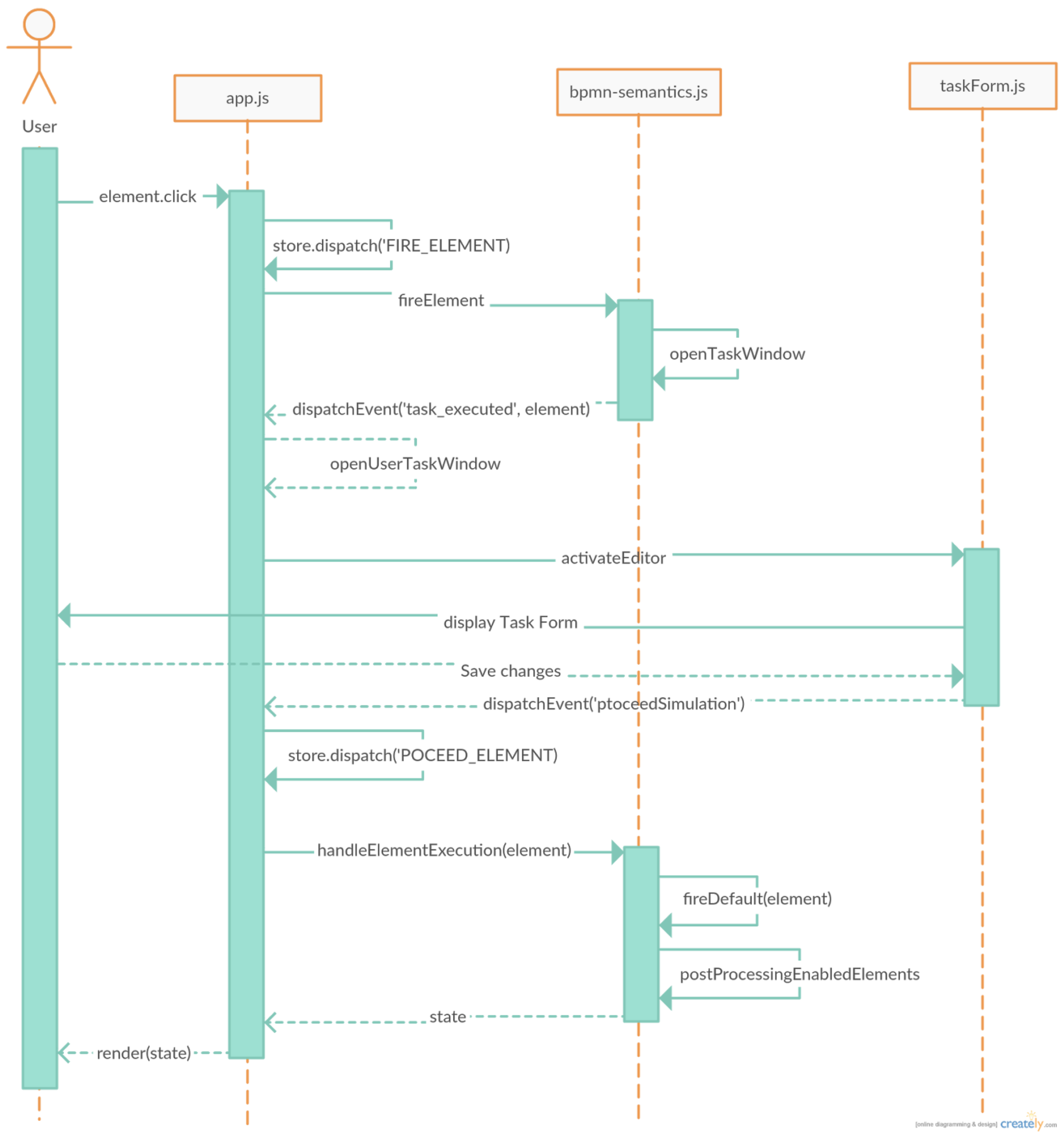

User
app.js
bpmn-semantics.js
taskForm.js
element.click
store.dispatch('FIRE_ELEMENT)
fireElement
openTaskWindow
dispatchEvent('task_executed', element)
openUserTaskWindow
activateEditor
display Task Form
Save changes
dispatchEvent('ptoceedSimulation')
store.dispatch('POCEED_ELEMENT)
handleElementExecution(element)
fireDefault(element)
postProcessingEnabledElements
state
render(state)

Figure 18: Execution of User Task

The second scenario presented is the execution of a script task, when the user executes the next step of the animation and the first enabled element in the queue is a script task. The execution can be seen in figure 19. We can observe that in this case, no visual form is implicated in the execution process, just that the script attached to the task is executed before the new state is returned.

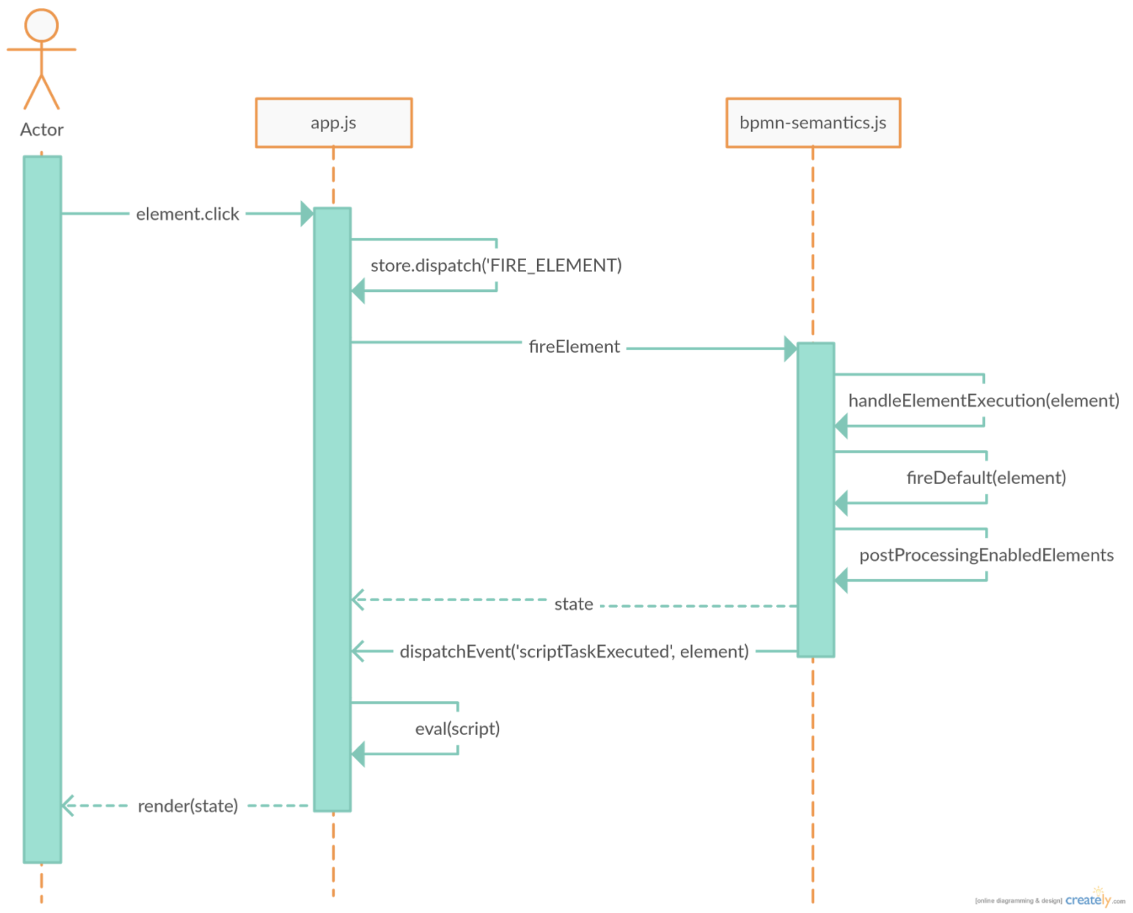

Actor
app.js
bpmn-semantics.js
element.click
store.dispatch('FIRE_ELEMENT)
fireElement
handleElementExecution(element)
fireDefault(element)
postProcessingEnabledElements
state
dispatchEvent('scriptTaskExecuted', element)
eval(script)
render(state)
creately

Figure 19: Execution of Script Task

The next scenario of interest is represented by the execution of the event-based split gateway, which will leave its successors event elements in a selectable state. The scenario is pretty similar to the previous two, with the difference that the first element to be executed is represented by the gateway. The execution sequence can be seen in figure 20. In this case the element is executed without any special events or visual forms involved. The only substantial difference is represented by the execution of the gateway element, handled in the *fireEventSplit* method.

The last scenario whose behaviour will be presented is represented by the explicit selection by the user of a boundary event element, when this is in a selectable state along with the task to which it is attached. In this case, the user makes a specific choice as to which element will be executed next, unlike the previous scenarios, when he would just select the action to advance the animation. The behaviour is represented in figure 21. The boundary event is treated as any other element during the execution, due to the fact that all the specific behaviour of the element is encapsulated in the *selectElement* method.

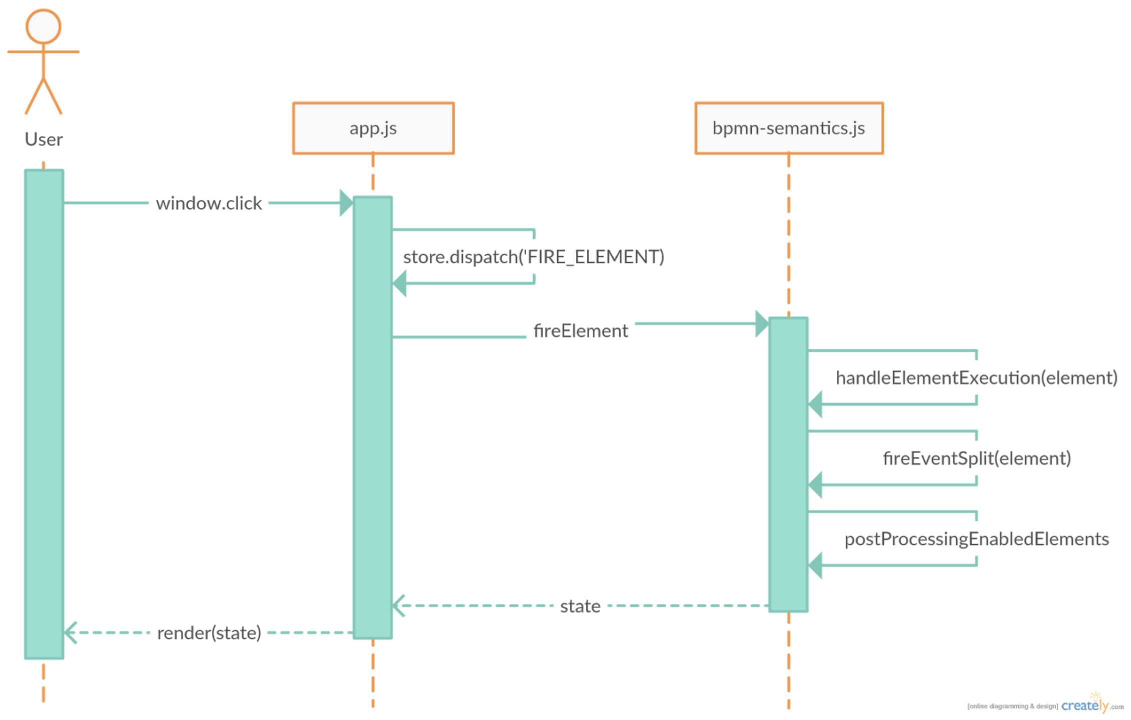
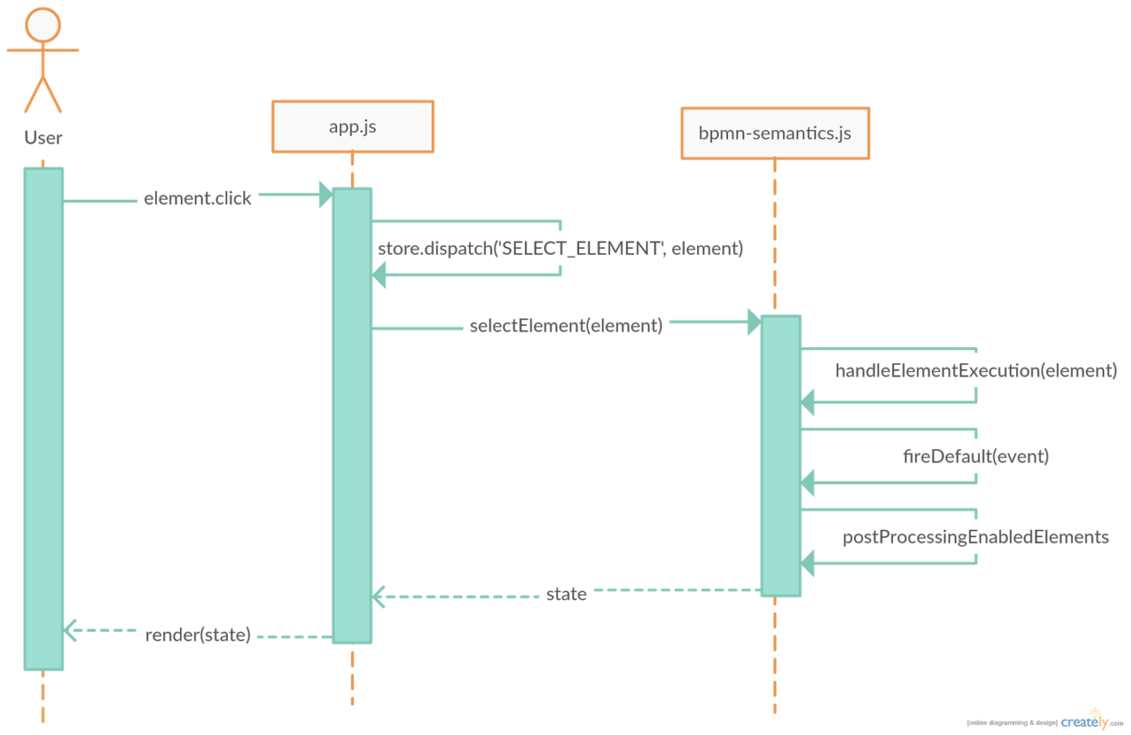


Figure 20: Execution of an Event-based Split Gateway



User
app.js
bpmn-semantics.js
element.click
store.dispatch('SELECT_ELEMENT', element)
selectElement(element)
handleElementExecution(element)
fireDefault(event)
postProcessingEnabledElements
state
render(state)

Figure 21: Selection of a Boundary Event

As it can be noticed from the sequence diagrams, the functional handling of the BPMN elements is done in the *bpmn-semantics* script, while the app script is responsible for opening the forms and rendering the visual state of the process.

3.3.1.2.2 Distributed Implementation

As in the case of the single-user version, the first scenario presented is the execution of the user task as part of an animation step. The behaviour can be seen in figure 22. As it can be seen in the diagram, the main difference from the single-user version of the animation is that the element type test is done at the app.js script level in order to reduce the number of messages exchanged between the two applications. The server application becomes aware of the task execution, only after the user closes the form, unlike the former implementation, when the element type test was done at the semantics level. Apart from this, the execution flow of the two versions is quite similar, the same methods being called in both cases.

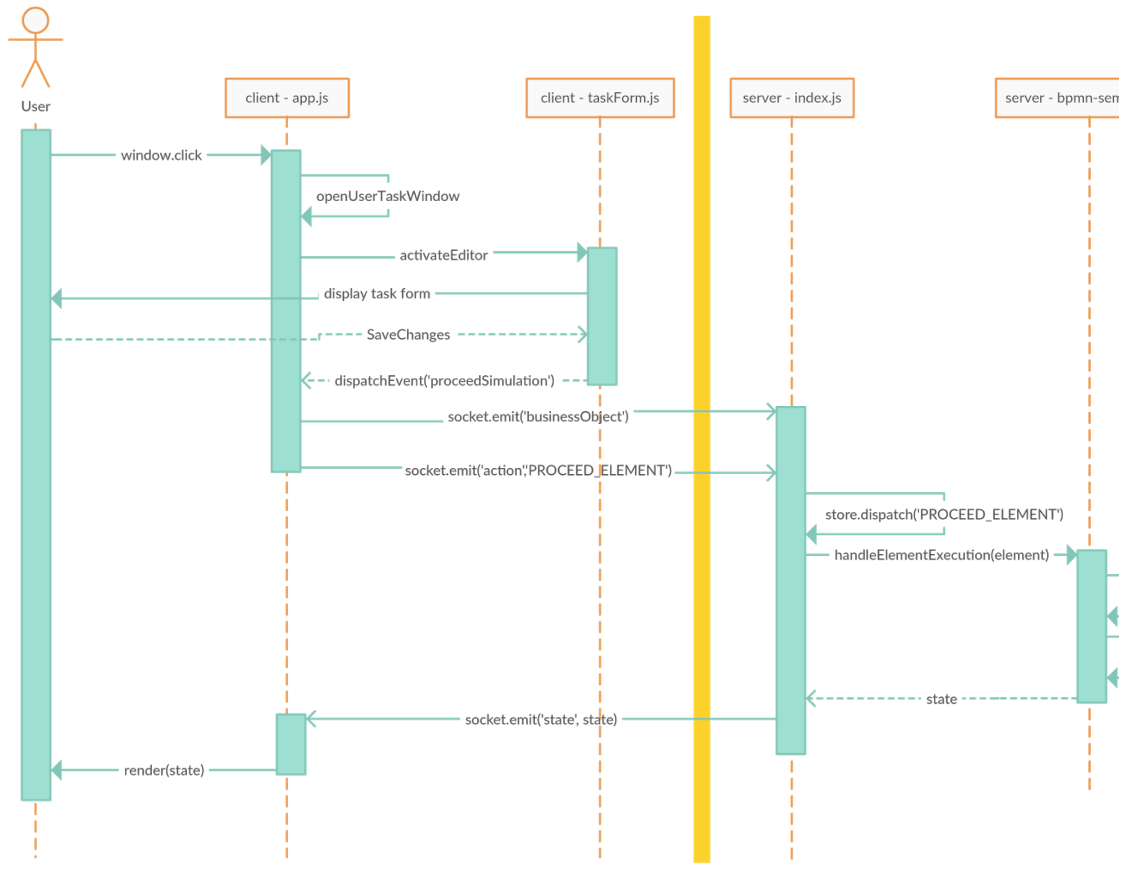

User
client - app.js
client - taskForm.js
server - index.js
server - bpmn-sem
window.click
openUserTaskWindow
activateEditor
display task form
SaveChanges
dispatchEvent('proceedSimulation')
socket.emit('businessObject')
socket.emit('action','PROCEED_ELEMENT')
store.dispatch('PROCEED_ELEMENT')
handleElementExecution(element)
state
socket.emit('state', state)
render(state)

Figure 22: Execution of a User Task

The next scenario can be seen in figure 23, and it represents the execution of a script task using the multi-user, distributed simulator. The main difference from the centralized execution of the same element is that the execution of the script attached to the task is performed on the client side and afterwards the modified business data is sent to the server.

The third diagram, which can be seen in figure 24 presents the execution of an event-based gateway. Since no special additional functionality is attached to this element, only the BPMN functional behaviour has to be handled by the simulator. It can be noticed that the behaviour flow is quite similar to the centralized implementation, the only new element being the client-server communication mechanism.

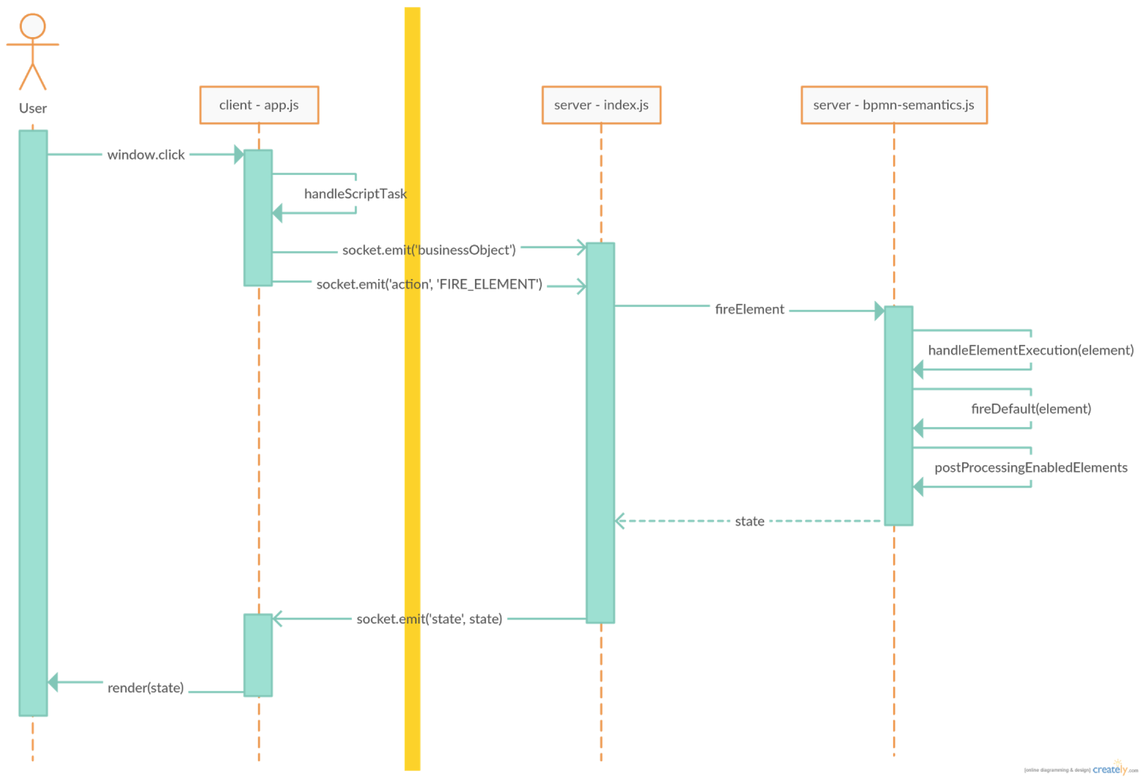

User
client - app.js
server - index.js
server - bpmn-semantics.js
window.click
handleScriptTask
socket.emit('businessObject')
socket.emit('action', 'FIRE_ELEMENT')
fireElement
handleElementExecution(element)
fireDefault(element)
postProcessingEnabledElements
state
socket.emit('state', state)
render(state)

Figure 23: Execution of a Script Task

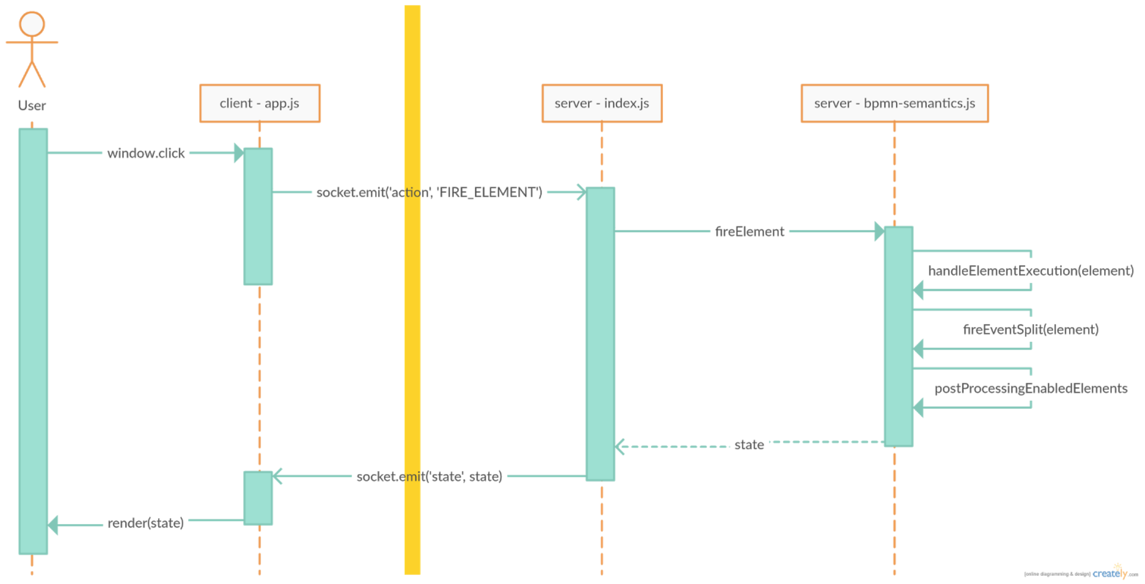

User
client - app.js
server - index.js
server - bpmn-semantics.js
window.click
socket.emit('action', 'FIRE_ELEMENT')
fireElement
handleElementExecution(element)
fireEventSplit(element)
postProcessingEnabledElements
state
socket.emit('state', state)
render(state)

Figure 24: Execution of an Event-based Split Gateway

The last behavioural scenario presented for the distributed tool is, as in the case of the single-user version, the selection of a boundary event by the user (figure 25). As in the previous case of the event-based gateway, the behaviour is almost identical to the non-distributed version, with one exception: the presence of the client-server exchanged messages.

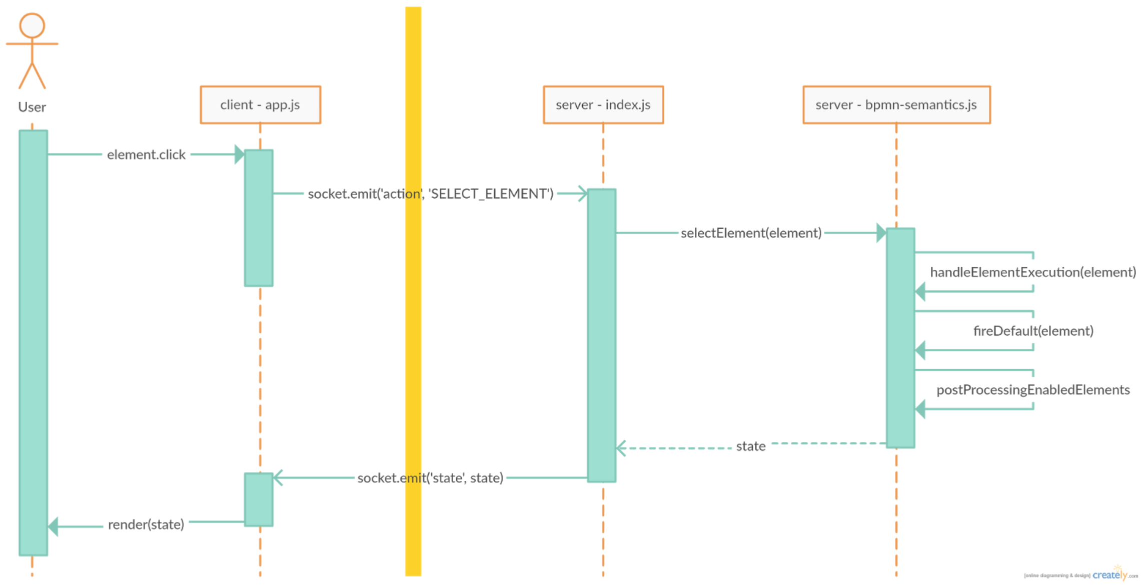
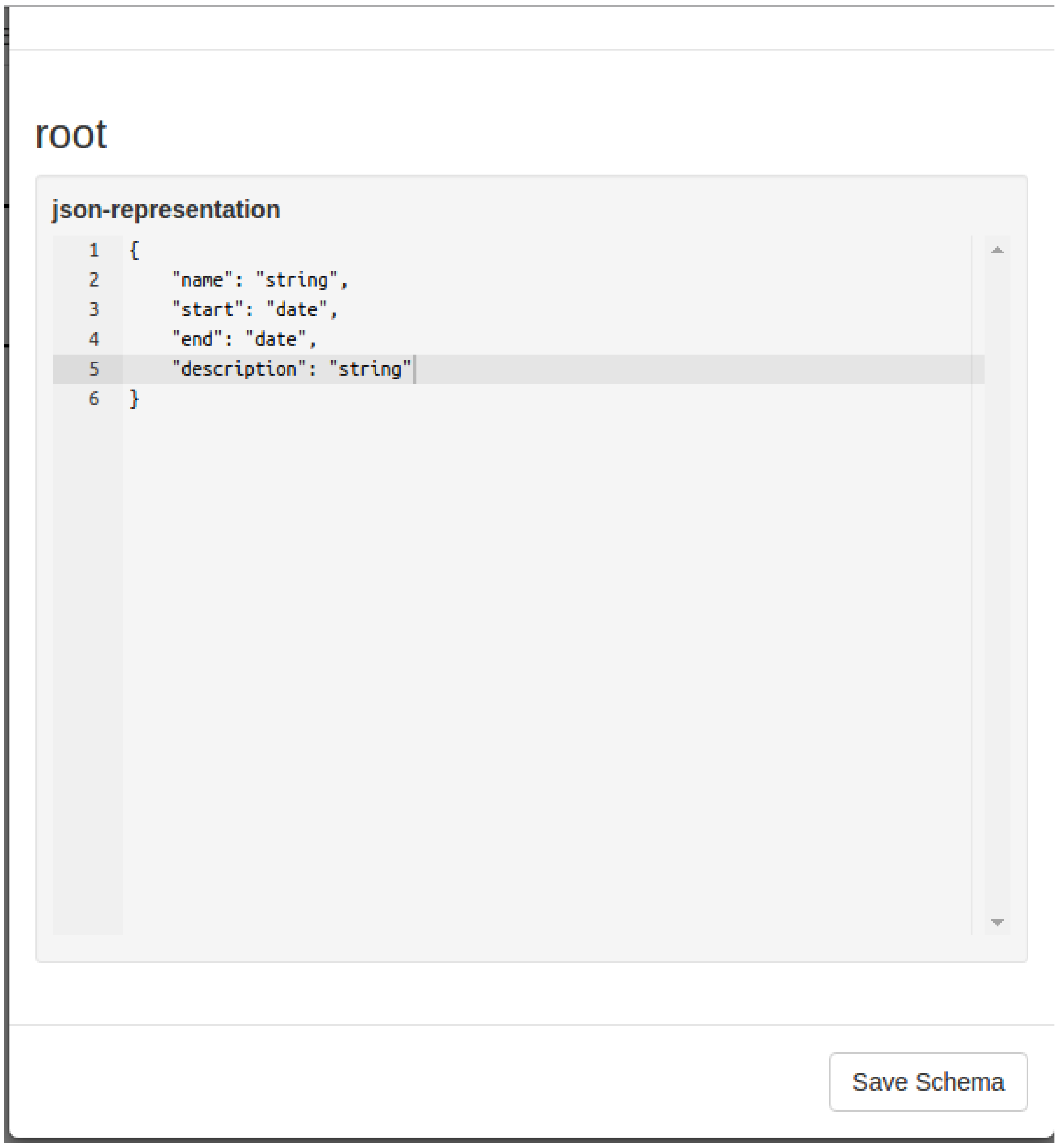

User
client - app.js
server - index.js
server - bpmn-semantics.js
element.click
socket.emit('action', 'SELECT_ELEMENT')
selectElement(element)
handleElementExecution(element)
fireDefault(element)
postProcessingEnabledElements
state
socket.emit('state', state)
render(state)

Figure 25: Selection of a boundary Event

The four execution scenarios presented in this section were chosen in order to cover a wide range of the functionality provided by the simulator, such as: display of a form, execution of an element, or selection of a selectable element, and to show the way in which some representative elements of the BPMN modeling, belonging to different category types, are handled during the animation.

3.3.2 Forms

In this section all the visual forms used in the application will be discussed in details and explanations will be provided as to their purpose and behaviour. The simulator uses visual forms in order to allow the user to provide input data during both, the editing and the animation phase. It is worth mentioning that, with the exception of the form used to define the global schema, each type of form is attached to a specific type of BPMN element and that the form will be displayed when the element is clicked by the user during the editing phase, or when the element is executed during the animation, depending on the purpose of the form. The forms are displayed on the screen as modal popup dialogs, which the user has to close prior to continuing with the animation. All the forms are implemented using the ***json-editor.js* library** [28], which receives as input a JSON schema and creates an HTML form based on the properties of the schema. During the development process, another similar technology was tested and considered for implementing the forms, named: *backbone-forms* library [29]. However, *json-editor* was preferred over the other option due to the simplicity of use and low learning curve. Each form has an editor object attached to it, where the values for all the form properties are maintained and can be accessed from the code. When the form is displayed its input fields will be filled-in with the values hold by the editor object. The application's forms can be divided in two large categories: forms which are used during the editing phase, in order to allow the user to define the global business data schema and the BPMN data definitions for each element, and the forms displayed during the animation, which allow the user to modify the content of the business data. In the rest of the section, each individual form will be presented along with a visual example.

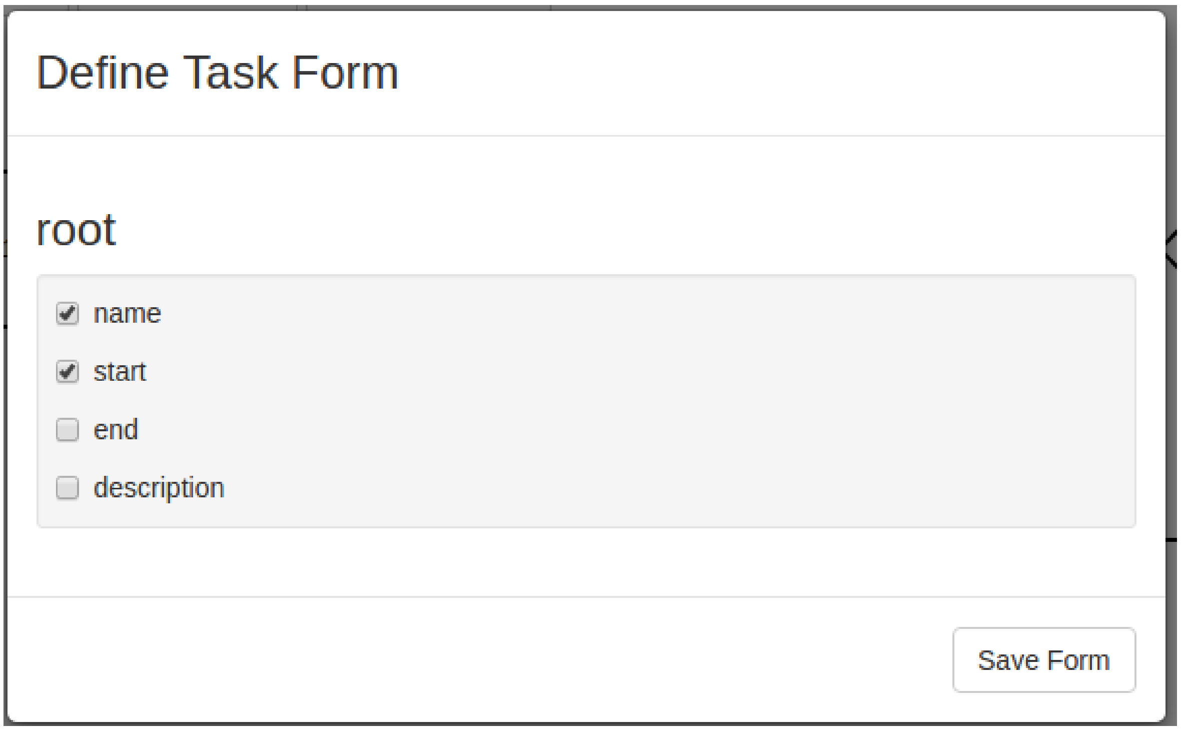

root

```
{
    "name": "string",
    "start": "date",
    "end": "date",
    "description": "string"
}
```

Figure 26: Define Business Data Schema Form

The first presented form is used during the editing phase for **defining the schema of the global business data** of the process (figure 26), and is defined in the ***taskSchemaForm.js*** script file. The form is displayed when the user clicks the “Define Task Schema” button, and it contains only one input field, where the user can define the JSON schema, based on which the business data object will be created. The schema consists of pairs of names and types which will define the properties of the process data. The business schema can only be modified during the editing phase, and every time a new schema is saved by the user, the list of modifiable properties attached to tasks and intermediary events will be reseted.

During the editing phase, there are multiple elements which have forms attached in order to allow the user to define some element specific information which will be used during the animation. The first such form is represented by the **user task form for defining the business data properties which can be modified** during the animation when the task is executed (figure 27) and is defined in the ***taskActivationForm.js***. The input fields of the form are of type boolean and they are defined based on the previously introduced schema of the business data. Therefore, the name of each defined business property will be displayed as a checkbox, and the selected ones will represent the properties to which the users have access, during the animation phase, when the element is executed. The form is displayed during the editing phase, when the user clicks on a user task element.


Define Task Form

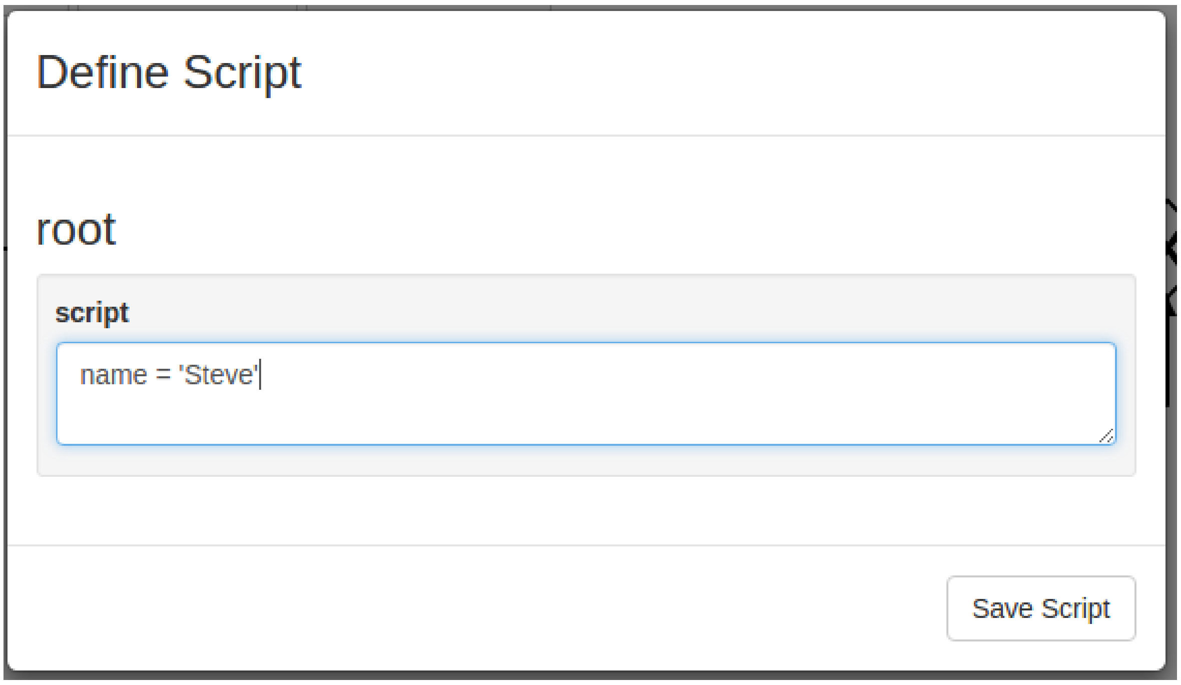
root

[✓] name
[✓] start
[ ] end
[ ] description

Figure 27: Edit User Task Properties Form

The second type of form used during the editing phase is the very similar to the user task form, but it is attached to **intermediate message events, for defining the properties which can be modified** when the event is executed, and it is defined in the ***messageEventForm.js*** script. The form works in the same way as the previously described one, with the exception that it is attached to intermediate message events. From a visual point of view, the form looks exactly like the one shown in figure 27.

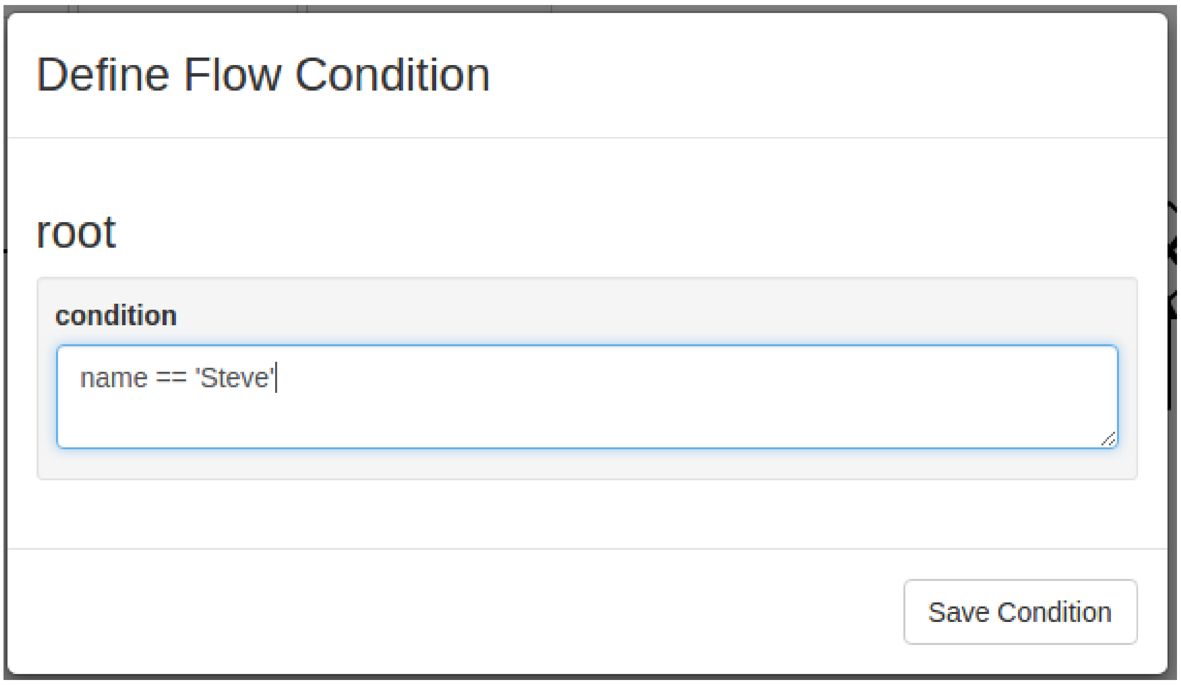
The next type of element which has a form attached during the editing phase is the script task. The form is used in order to **define the script which will be executed along with the task during the animation** (figure 28), and it is defined in the ***scriptForm.js*** script. The form contains only one input, where the user has to define the script as a text. Although the script is also defined based on the properties of the global business data, there is no explicit test of the validity of the script, until the moment when the script is actually executed. Therefore, it is the responsibility of the user to use only the properties defined in the business data in order to provide a valid script. The form is only displayed during the editing phase, when a user clicks on a script task.



name = 'Steve'

Figure 28: Form for defining a Script in a Script Task

The last form which can be used during the editing phase is represented by the **sequence flow condition definition form** (figure 29), which is defined in the ***flowForm.js*** script. The form is opened during the editing phase when the user clicks on a sequence flow in the process. Although all edges can have conditions defined, such conditions are only relevant when the edge is placed after a XOR split gateway. In this situation, if one of the conditions attached to an edges which follows the gateway, proves to be true, the path of the execution flow after the gateway, can be determined automatically, without the necessity for the user to select it manually, during the animation. The form has only one input, which is used to define the condition in the form of text. As in the case of the script definition, the user has to pay attention to use only the properties defined in the global business data, in order for the condition to be evaluated.


Define Flow Condition

root

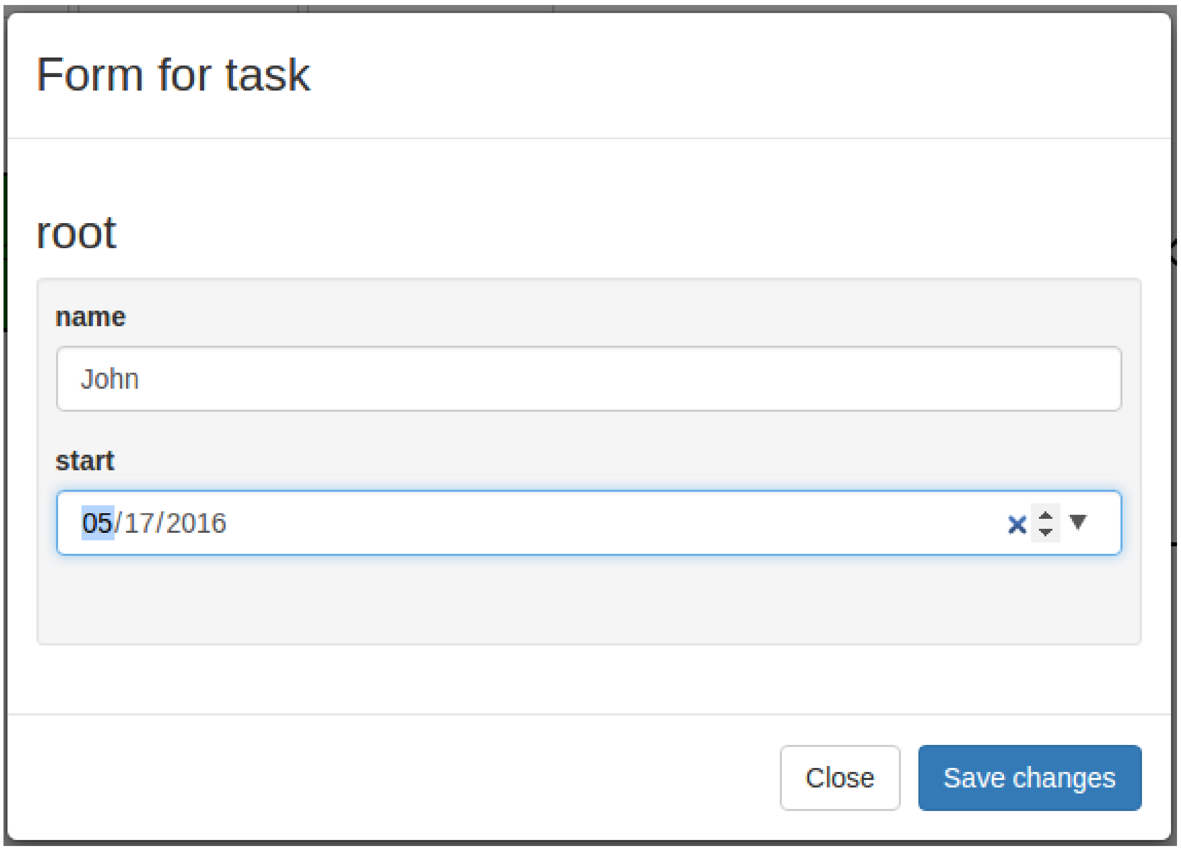
condition

name == 'Steve'

Figure 29: Form for defining Sequence Flow Conditions

In addition to the forms used during the editing phase for defining the data content of the process during the animation, there is also a set of forms which are used during the animation to actually interact with the process data values. The first such form is the one **attached to a user task, which allow the user to set values for the business properties** which are editable upon the task's execution, and is defined in the ***taskForm.js*** file. Intuitively, this form will be opened before a user task is executed during the simulation. When the form is displayed, the user can decide to close the form without executing the task, or to proceed with the task execution. The content of the form depends on the properties of the business data, and the ones which were set as editable for the current task. Each type of property is displayed as a different type of input (string as text input, date as calendar, etc.). In figure 30, the content has been defined based on the selected properties in figure 27. Inside this form, the user can modify the values of each property displayed, and the new values will be stored in the global business data.

The form which is **attached to the intermediate message event for allowing the user to modify the values of the business properties** which are editable when the event arrives looks exactly like the form attached to the user tasks during the animation (figure 30), and is defined in the ***eventMessage.js*** script. The functional purpose of the form is the same as the one attached to user tasks, but however the moment during the animation when it is displayed is a bit different. If in the case of the user task form, the form is displayed before the user task is executed, while the intermediate message event form is displayed after the element to which it is attached is executed during the animation. The reason for this different behaviour is that messages are not executed whenever the user decides, but rather they are being sent by a third party entity. Often, such messages are being accompanied by data, therefore the form mechanism is used to simulate the arrival of data, by allowing the user to input it himself.


Form for task

root

name

John

start

05/17/2016

Figure 29: Define values for business properties

The last modal window used during the animation phase of the simulator is the **enablement form**, which is defined in the ***enablementForm.js*** script, and can be seen in figure 31. Unlike the previous presented forms, this window can be attached to any type of element, but only when the element is the successor of an OR split gateway. In addition, this window does not contain a form, its only purpose being to allow the user to decide if he wants to enable or disable the path represented by the current BPMN element. This is necessary due to the fact that after an OR split gateway, a flowing path may be executed or not. Therefore, all the successors of such a gateway will display upon their selection, this dialog in order to allow the user to decide.

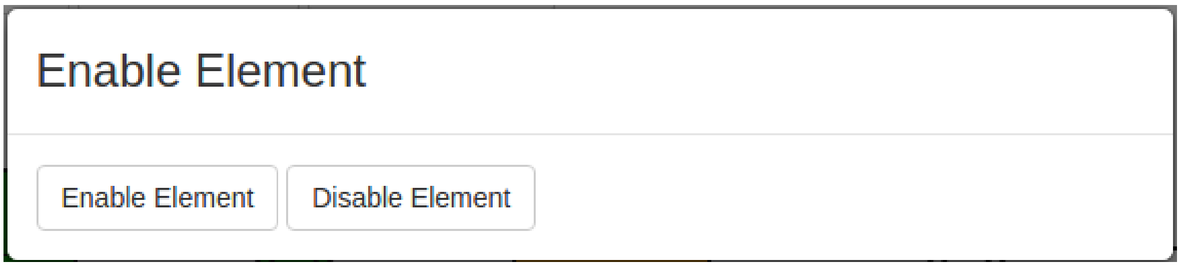

Enable Element

Figure 31: Enablement Form after an OR gateway

The forms play a very important role both before and during the animation, by providing the user with the opportunity to define the business data, and to modify its values during the execution of the process.

3.3.3 Execution State

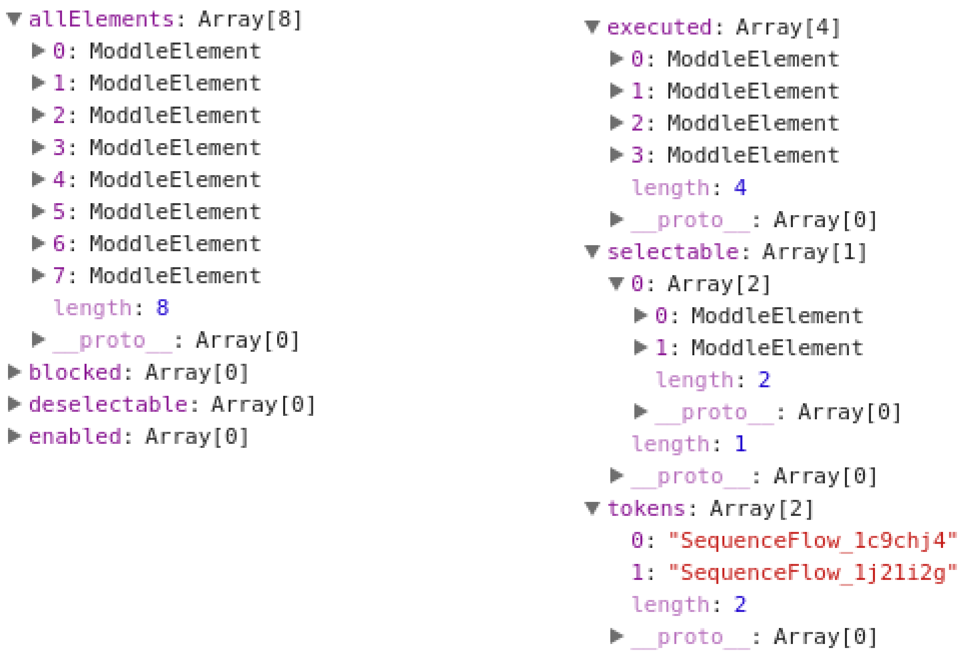
The content of this subsection dwells upon the internal structure of the process state and the way in which the process evolves during its execution by changing its current state. In the first part of the subsection, the purpose of the process state will be defined and each property which is part of the state will be described in details. In the second part, the way in which the process change its current state upon the execution of an element will be explained, and a relevant example will be presented for this purpose.

3.3.3.1 State Description

On a theoretical level, the state of the entire process is composed from the individual states of all the elements of process and the placement of tokens on the edges of the process. The states of an individual element have been presented in section 3.2.2, while the concept of "token" has been introduced in section 3.2.3. Whenever an element is executed in the process, a new process state is created, based on the previous process state and the behaviour of the element being executed.

From a technical point of view, the state of the process is represented as an object containing a number of element arrays as properties. An example of the internal representation of the state object used in the implementation of the simulator can be seen in figure 32. In what follows, each property of the process state would be described individually, along with its purpose.

The ***allElements*** property represents an array in which all the flow elements of the process are placed. The array is used in the beginning of the simulation when the data definitions are defined for each type of element, but the most important usage is the ability to parse through all the element of the process in order to remove the visual markers of the old state, when the new state of the process is visually rendered.



```
▼allElements: Array[8]
  ▶0: ModdleElement
  ▶1: ModdleElement
  ▶2: ModdleElement
  ▶3: ModdleElement
  ▶4: ModdleElement
  ▶5: ModdleElement
  ▶6: ModdleElement
  ▶7: ModdleElement
   length: 8
  ▶__proto__: Array[0]
▶blocked: Array[0]
▶deselectable: Array[0]
▶enabled: Array[0]
```

```
▼executed: Array[4]
  ▶0: ModdleElement
  ▶1: ModdleElement
  ▶2: ModdleElement
  ▶3: ModdleElement
   length: 4
  ▶__proto__: Array[0]
▼selectable: Array[1]
  ▼0: Array[2]
    ▶0: ModdleElement
    ▶1: ModdleElement
     length: 2
    ▶__proto__: Array[0]
   length: 1
  ▶__proto__: Array[0]
▼tokens: Array[2]
   0: "SequenceFlow_1c9chj4"
   1: "SequenceFlow_1j21i2g"
   length: 2
  ▶__proto__: Array[0]
```

Figure 32: Internal Representation of the Process State

The next property of the state object displayed in figure 32 is called ***blocked*** and it is an array which contains all the elements which are in a blocked state in the current state of the process. The array is used upon state rendering in order to display the specific visual marker of the blocked element state for all the elements in the array.

The ***deselectable*** property is also an array which contains the elements placed in a selectable state, but for which the user has the option to deselect them and return them to an unexecuted state. This scenario happens with the elements that are successors of an OR split gateway. Although during the rendering, these elements are given the selectable visual marker, they are kept separately in order to be able to display the enablement window when they are selected.

The ***enabled*** array property holds the elements which are enabled during the current state of the process. Upon state rendering, all these elements receive the enabled marker. When the user clicks inside the diagram view the first element from the array will be executed, and the new process state will be created. In the above example, the array is empty, which means that the only option available to the user, at this point, is to select one selectable element in order to continue with the animation.

The ***executed*** array contains all the elements which were already executed during the process animation. Like the other properties, the array is used for visual rendering, in order to assign the executed marker to all the executed elements.

The ***selectable*** property is a special type of array, in the sense that its elements are also arrays, unlike the previous properties, where each element of the array was a BPMN element. This array contains groups of elements which are placed into a selectable state. The unusual structure of the array is motivated by the need to deselect automatically the other elements in the group after one element in the same group have been selected by the user. The most common example is represented by the successors elements of a XOR split gateway. Upon the selection of one successor by the user, all the others have to be deselected automatically by the simulator. By implementing this custom type of structure, the access to all the elements which need to be deselected is done very fast and in straightforward way.

The last array property of the state object is represented by the ***tokens*** array. Unlike the previous arrays which contain BPMN objects as elements, this array contains only the IDs of the sequence flow objects on which the tokens are placed in the current state of the process. The tokens are used in order to determine the enablement of the process elements, especially the OR join gateway, whose enablement requires a more complex computation.

The only difference between the state representation used by the single-user version and the one used by the distributed version of the simulator is that the latter also contains a boolean property called ***started***. This property is necessary in the distributed environment in order to allow the clients to determine if the animation is already started, upon receiving the current state from the server side. The most relevant example is the state right after one of the clients starts the simulation, by pressing the start button. In this case, no element is executed straight away, so upon receiving the new state, the other clients would not be able to determine that the simulation have been started, due to the fact that the state would look exactly the same. By introducing this boolean property, the server can assure the full synchronization of the clients, even in the case when a client connects latter to the process, when the animation is already started.

3.3.3.2 Execution State Change

Upon the execution of an element in the process, the overall internal state of the process has to be updated in order to reflect the new situation into which the process evolved. In order to compute a new state of the process, the simulator uses the current state and the element of the process which is being executed. Based on the type of the element, on its successor elements, and also on its incoming connections, the content of the properties of the new state is determined. In what follows a concrete example will be presented and explained in order to understand how the process state can evolve during the process execution.

The process to be executed is shown in figure 33, while the internal representation of the initial state can be observed in figure 34. Only the states representing to the first four steps of the process execution will be presented, in this section in order to form a clear idea of how the state transition is performed upon the execution of different types of elements. For the sake of simplicity and in order to reduce the space occupied by the figures representing the states, all the irrelevant data, such as element data definitions, have been removed from the representation of the elements.

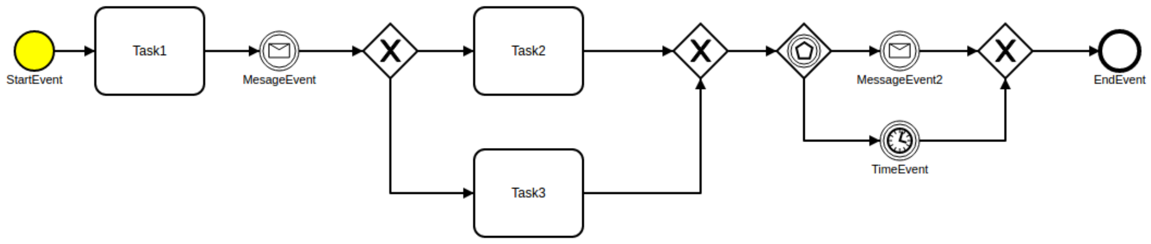


Figure 33: Executed Process

As it can be seen in figure 34, in the initial state, only the start event of the process can be found in the *enabled* array, while the *allElements* array contains all the 12 elements of the process. The rest of the arrays are empty due to the fact that no element have been executed yet, such that some of the elements of the process would be placed in the states corresponding to these arrays. In addition, the *tokens* array is also empty, since there are no tokens in the process yet.

```
{
 "allElements": [
   {
     "$type": "bpmn:StartEvent",
     "id": "StartEvent_1",
     "name": "StartEvent"
   },
   {
     "$type": "bpmn:Task",
     "id": "Task_0u8sui7",
     "name": "Task1"
   },
   {
     "$type": "bpmn:IntermediateCatchEvent",
     "id": "IntermediateCatchEvent_1gjxev4",
     "name": "MesageEvent"
   },
   {
     "$type": "bpmn:ExclusiveGateway",
     "id": "ExclusiveGateway_106l8ay"
   },
   {
     "$type": "bpmn:Task",
```

```
{
  "$type": "bpmn:EventBasedGateway",
  "id": "EventBasedGateway_121t13j"
},
{
  "$type": "bpmn:IntermediateCatchEvent",
  "id": "IntermediateCatchEvent_04tom7u",
  "name": "TimeEvent"
},
{
  "$type": "bpmn:IntermediateCatchEvent",
  "id": "IntermediateCatchEvent_0xjam1l",
  "name": "MessageEvent2"
},
{
  "$type": "bpmn:ExclusiveGateway",
  "id": "ExclusiveGateway_0kd1m7k"
},
{
  "$type": "bpmn:EndEvent",
  "id": "EndEvent_0egttjp",
  "name": "EndEvent"
}
```

```
      "id": "Task_1rtqxpe",
      "name": "Task2"
    },
    {
      "$type": "bpmn:Task",
      "id": "Task_14xah4d",
      "name": "Task3"
    },
    {
      "$type": "bpmn:ExclusiveGateway",
      "id": "ExclusiveGateway_18btlo4"
    },
```
```
  ],
  "executed": [],
  "enabled": [
    {
      "$type": "bpmn:StartEvent",
      "id": "StartEvent_1",
      "name": "StartEvent"
    }
  ],
  "selectable": [],
  "tokens": [],
  "blocked": [],
  "deselectable": []
}
```

Figure 34: State 1 of the Process Execution

Once the start event is executed, the new state of the process can be seen in figure 35. Now, one can notice that the start event appears in the *executed* array, while the following task have been placed in the *enabled* state. Now the *tokens* array contains the token created by the execution of the start event.

```
{
  "allElements": [
    {
      "$type": "bpmn:StartEvent",
      "id": "StartEvent_1",
      "name": "StartEvent"
    },
    {
      "$type": "bpmn:Task",
      "id": "Task_0u8sui7",
      "name": "Task1"
    },
    {
      "$type": "bpmn:IntermediateCatchEvent",
      "id": "IntermediateCatchEvent_1gjxev4",
      "name": "MesageEvent"
    },
    {
      "$type": "bpmn:ExclusiveGateway",
      "id": "ExclusiveGateway_106l8ay"
    },
    {
      "$type": "bpmn:Task",
      "id": "Task_1rtqxpe",
      "name": "Task2"
    },
    {
      "$type": "bpmn:Task",
      "id": "Task_14xah4d",
      "name": "Task3"
    },
    {
      "$type": "bpmn:ExclusiveGateway",
      "id": "ExclusiveGateway_18btlo4"
    },
    {
      "$type": "bpmn:EventBasedGateway",
      "id": "EventBasedGateway_121t13j"
    },
```
```
    {
      "$type": "bpmn:IntermediateCatchEvent",
      "id": "IntermediateCatchEvent_04tom7u",
      "name": "TimeEvent"
    },
    {
      "$type": "bpmn:IntermediateCatchEvent",
      "id": "IntermediateCatchEvent_0xjam1l",
      "name": "MessageEvent2"
    },
    {
      "$type": "bpmn:ExclusiveGateway",
      "id": "ExclusiveGateway_0kd1m7k"
    },
    {
      "$type": "bpmn:EndEvent",
      "id": "EndEvent_0egttjp",
      "name": "EndEvent"
    }
  ],
  "executed": [
    {
      "$type": "bpmn:StartEvent",
      "id": "StartEvent_1",
      "name": "StartEvent"
    }
  ],
  "enabled": [
    {
      "$type": "bpmn:Task",
      "id": "Task_0u8sui7",
      "name": "Task1"
    }
  ],
  "selectable": [],
  "tokens": [
    "SequenceFlow_1928jq7"
  ],
  "blocked": [],
  "deselectable": []
}
```

Figure 35: State 2 of the Process Execution

After *Task1* is executed the next message event is placed in a selectable state. This time the *enabled* array is empty and the user has to make a manual selection in order to continue with the animation. The event is placed alone in the group, due to the fact that its selection will not deactivate any other elements of the process. The execution of the task consumed the token from the previous state and created a new one on the next edge. The new process state can be seen in figure 36.

```
{
 "allElements": [
   {
    "$type": "bpmn:StartEvent",
    "id": "StartEvent_1",
    "name": "StartEvent"
   },
   {
    "$type": "bpmn:Task",
    "id": "Task_0u8sui7",
    "name": "Task1"
   },
   {
    "$type": "bpmn:IntermediateCatchEvent",
    "id": "IntermediateCatchEvent_1gjxev4",
    "name": "MesageEvent"
   },
   {
    "$type": "bpmn:ExclusiveGateway",
    "id": "ExclusiveGateway_106l8ay"
   },
   {
    "$type": "bpmn:Task",
    "id": "Task_1rtqxpe",
    "name": "Task2"
   },
   {
    "$type": "bpmn:Task",
    "id": "Task_14xah4d",
    "name": "Task3"
   },
   {
    "$type": "bpmn:ExclusiveGateway",
    "id": "ExclusiveGateway_18btlo4"
   },
   {
    "$type": "bpmn:EventBasedGateway",
    "id": "EventBasedGateway_121t13j"
   },
   {
    "$type": "bpmn:IntermediateCatchEvent",
    "id": "IntermediateCatchEvent_04tom7u",
    "name": "TimeEvent"
   },
   {
    "$type": "bpmn:IntermediateCatchEvent",
    "id": "IntermediateCatchEvent_0xjam1l",
    "name": "MessageEvent2"
   },
   {
    "$type": "bpmn:ExclusiveGateway",
    "id": "ExclusiveGateway_0kd1m7k"
   },
   {
    "$type": "bpmn:EndEvent",
    "id": "EndEvent_0egttjp",
    "name": "EndEvent"
   }
 ],
 "executed": [
   {
    "$type": "bpmn:Task",
    "id": "Task_0u8sui7",
    "name": "Task1"
   },
   {
    "$type": "bpmn:StartEvent",
    "id": "StartEvent_1",
    "name": "StartEvent"
   }
 ],
 "enabled": [],
 "selectable": [
   [
    {
     "$type": "bpmn:IntermediateCatchEvent",
     "id": "IntermediateCatchEvent_1gjxev4",
     "name": "MesageEvent"
    }
   ]
 ],
 "tokens": [
   "SequenceFlow_1ah8705"
 ],
 "blocked": [],
 "deselectable": []
}
```

Figure 36: State 3 of the Process Execution

With the selection of the message event, the process animation advances to a new state (figure 37). Here the message event is transferred to the *executed* elements, while the XOR gateway, which represents its successor is *enabled*.

```
{
 "allElements": [
   {
    "$type": "bpmn:StartEvent",
    "id": "StartEvent_1",
    "name": "StartEvent"
   },
   {
    "$type": "bpmn:Task",
    "id": "Task_0u8sui7",
    "name": "Task1"
   },
   {
```
```
   {
    "$type": "bpmn:IntermediateCatchEvent",
    "id": "IntermediateCatchEvent_0xjam1l",
    "name": "MessageEvent2"
   },
   {
    "$type": "bpmn:ExclusiveGateway",
    "id": "ExclusiveGateway_0kd1m7k"
   },
   {
    "$type": "bpmn:EndEvent",
    "id": "EndEvent_0egttjp",
    "name": "EndEvent"
    "$type": "bpmn:IntermediateCatchEvent",
    "id": "IntermediateCatchEvent_1gjxev4",
    "name": "MesageEvent"
  },
  {
    "$type": "bpmn:ExclusiveGateway",
    "id": "ExclusiveGateway_106l8ay"
  },
  {
    "$type": "bpmn:Task",
    "id": "Task_1rtqxpe",
    "name": "Task2"
  },
  {
    "$type": "bpmn:Task",
    "id": "Task_14xah4d",
    "name": "Task3"
  },
  {
    "$type": "bpmn:ExclusiveGateway",
    "id": "ExclusiveGateway_18btlo4"
  },
  {
    "$type": "bpmn:EventBasedGateway",
    "id": "EventBasedGateway_121t13j"
  },
  {
    "$type": "bpmn:IntermediateCatchEvent",
    "id": "IntermediateCatchEvent_04tom7u",
    "name": "TimeEvent"
  },
  }
 ],
 "executed": [
  {
    "$type": "bpmn:IntermediateCatchEvent",
    "id": "IntermediateCatchEvent_1gjxev4",
    "name": "MesageEvent"
  },
  {
    "$type": "bpmn:Task",
    "id": "Task_0u8sui7",
    "name": "Task1"
  },
  {
    "$type": "bpmn:StartEvent",
    "id": "StartEvent_1",
    "name": "StartEvent"
  }
 ],
 "enabled": [
  {
    "$type": "bpmn:ExclusiveGateway",
    "id": "ExclusiveGateway_106l8ay"
  }
 ],
 "selectable": [],
 "tokens": [
  "SequenceFlow_16d93vi"
 ],
 "blocked": [],
 "deselectable": []
}
```

Figure 37: State 4 of the Process Execution

Once the XOR split gateway is executed by the user, the two successor tasks are placed in the same group inside the *selectable* elements (figure 38), because no conditions are defined on the edges following the gateway. In this case the selection of one element will disable the other one. We can notice that in the *tokens* array there are two tokens for the first time, as the gateway has two successor elements.

```
{
 "allElements": [
  {
    "$type": "bpmn:StartEvent",
    "id": "StartEvent_1",
    "name": "StartEvent"
  },
  {
    "$type": "bpmn:Task",
    "id": "Task_0u8sui7",
    "name": "Task1"
  },
  {
    "$type": "bpmn:IntermediateCatchEvent",
    "id": "IntermediateCatchEvent_1gjxev4",
    "name": "MesageEvent"
  },
  {
    "$type": "bpmn:ExclusiveGateway",
    "id": "ExclusiveGateway_106l8ay"
  },
  {
    "$type": "bpmn:Task",
    "id": "Task_1rtqxpe",
    "name": "Task2"
  },
  {
    "$type": "bpmn:Task",
    "id": "Task_14xah4d",
  {
    "$type": "bpmn:EndEvent",
    "id": "EndEvent_0egttjp",
    "name": "EndEvent"
  }
 ],
 "executed": [
  {
    "$type": "bpmn:ExclusiveGateway",
    "id": "ExclusiveGateway_106l8ay"
  },
  {
    "$type": "bpmn:IntermediateCatchEvent",
    "id": "IntermediateCatchEvent_1gjxev4",
    "name": "MesageEvent"
  },
  {
    "$type": "bpmn:Task",
    "id": "Task_0u8sui7",
    "name": "Task1"
  },
  {
    "$type": "bpmn:StartEvent",
    "id": "StartEvent_1",
    "name": "StartEvent"
  }
 ],
 "enabled": [],
 "selectable": [
      "name": "Task3"
    },
    {
      "$type": "bpmn:ExclusiveGateway",
      "id": "ExclusiveGateway_18btlo4"
    },
    {
      "$type": "bpmn:EventBasedGateway",
      "id": "EventBasedGateway_121t13j"
    },
    {
      "$type": "bpmn:IntermediateCatchEvent",
      "id": "IntermediateCatchEvent_04tom7u",
      "name": "TimeEvent"
    },
    {
      "$type": "bpmn:IntermediateCatchEvent",
      "id": "IntermediateCatchEvent_0xjam1l",
      "name": "MessageEvent2"
    },
    {
      "$type": "bpmn:ExclusiveGateway",
      "id": "ExclusiveGateway_0kd1m7k"
    },
```

```
    [
      {
        "$type": "bpmn:Task",
        "id": "Task_1rtqxpe",
        "name": "Task2"
      },
      {
        "$type": "bpmn:Task",
        "id": "Task_14xah4d",
        "name": "Task3"
      }
    ]
  ],
  "tokens": [
    "SequenceFlow_14kzqhk",
    "SequenceFlow_0huztvp"
  ],
  "blocked": [],
  "deselectable": []
}
```

Figure 38: Step 5 of the Process Execution

The next steps of the process execution can be determined intuitively, in the same manner as the presented ones. Each change of the state of one element, determines the creation of a new overall state of the process.

3.4 Discussion

Although most of the decisions taken during the design or implementation phases of the simulator have been explained in the previous sections of the current chapter, there are several clarifications which can be made, regarding the implemented solution.

The first aspect which requires some clarifications is the decision to have two different versions of the simulator: the single-user and the multi-user. It is important to underline that both implementations provide some advantages and some inconveniences, and therefore the distributed version should not be seen as an improved version of the centralized one. The main advantage of this multi-user version is the ability to involve more than one user in one process execution, while the animation is executed in the exact same way like in the single-user version. However, this distributed implementation requires network traffic in order to exchange messages between the server and the client applications for maintaining the animation synchronized for all the clients. On the other hand, the single-user version has the limitation of allowing only one user to participate in a process animation, but it does not require any message exchange mechanism, which may be more suitable in certain scenarios. The two implementations serve the same purpose, but each one of them may be more suitable in certain situations, depending on the organization and process attributes.

Another important part of the application, which could benefit from some additional information, is represented by the states assigned to each individual BPMN element during the animation. The individual states which are used during the process execution, are not part of any BPM related standard, but rather they were created in order to accommodate all the possible situations in which an element can be found. Although the possible transitions from each state are rather straightforward, the situation becomes a bit more complicated when the executed process contains loops. In this situation, an element which was already executed, can become enabled or selectable again. Once an element is executed, it is never removed from the array which contains the executed elements. However, it can also be added to the arrays which correspond to the other states. When rendering the visual state of the process, the order in which the tool adds the state markers to the elements is predefined, such that the *executed* markers are added first. In this way, the enabled or selectable marker is added on top of the executed one, and when the element is no longer in an enabled or selectable state, the executed marker will be visible again.

4 Results and Case Study

The content of this chapter is devoted to analyzing the results achieved and compare them to the requirements of the tool, which were described in section 2.5. In the second part of the chapter, a case study will also be introduced, presenting step-by-step, how a real-life business process animation is executed by the system.

4.1 Achieved Results

By analyzing the functionality of the obtained tool, we can determine the degree of achievement of the requirements stated in the second chapter of this work. The first functional requirement was to assure the correct behaviour of the BPMN elements used for modelling the processes. In this sense, it can be said that the application supports a rather large number of different elements, and also that most of the core set elements of BPMN are supported by the application. The only major absence is represented by the ability to represent inter-organization collaborations, by using multiple pools in the process model. However, we consider that most business processes can be modelled and simulated using the supported collection of symbols. In addition, there are no known bugs in the behaviour of the BPMN elements implemented in the system.

The second functional requirement stated in section 2.5 was to implement a mechanism of window forms which would be used in order to allow the user to define and modify the business data attached to the process. We can say that the forms mechanism was implemented successfully in the simulator, allowing the user to define scripts, individual values for business properties, or flow conditions. In addition, it provides a flexible way to define the business data, with no other restrictions, than the type of each property. The type of a property has to be supported by the tool, such that the defined schema can be parsed in order to create customized forms during the animation phase.

The last major functional requirement was to provide a distributed environment for the process animation, such that more than one user could take part in the execution of a process. The application has indeed a distributed version, where multiple users can take part in both: defining the business data and executing the process. The implemented communication mechanism assures permanent synchronization between the different clients connected to the process execution, even in the case when a client connects later, during the animation or editing phase. As in the case of elements behaviour, there are no known bugs related to the distributed version communication or synchronization.

In terms of non-functional requirements, the tool can be considered a lightweight system, due to the lack of any installation and configuration requirements from the user. The tool is used in a browser window, therefore being accessible from any type of platform which supports a browser. In addition, the system can also be considered self-contained, due to the fact that there are no external dependencies required in order to run the animations.

The system testing has been done manually, using processes design especially with the purpose of covering all the supported elements, and testing some complicated design situations where bugs were more likely to appear. The diagrams were produced using the bpmn.io [30] modeller, while the testing have been done using the Chrome browser [31]. Each element has been tested in multiple process models, and whenever a potential special design context or exceptional behaviour could raise some execution problems, a special process diagram was designed, especially for the purpose of testing that the element behaves as expected in the given scenario.

4.2 Case Study

In this subsection, a case study will be presented in order to illustrate all the steps performed when executing a process animation. The selected process is adapted from [2] (page 325), by adding some BPMN elements which are supported by the tool, but were not present in the original process, such as intermediate events, in order to illustrate how these elements are handled by the system. In addition, the elements which were present in the original process, but are not supported by the simulator, have been removed, in order to be able to perform the process animation, using the tool.

The model describes an order fulfillment process in a company, which starts upon receiving a purchase order from a client. In the process, some exception scenarios are handled as well, along with the main success scenario which ends up with the shipment of the products and receivement of the payment. The process can be observed in figure 39, which shows the original state of the process, once it is uploaded in the simulator. The process is not meant to provide the best model design, due to the fact that it is used only for the purpose of demonstrating the capabilities of the simulator, and not for providing a good process model for the described scenario.

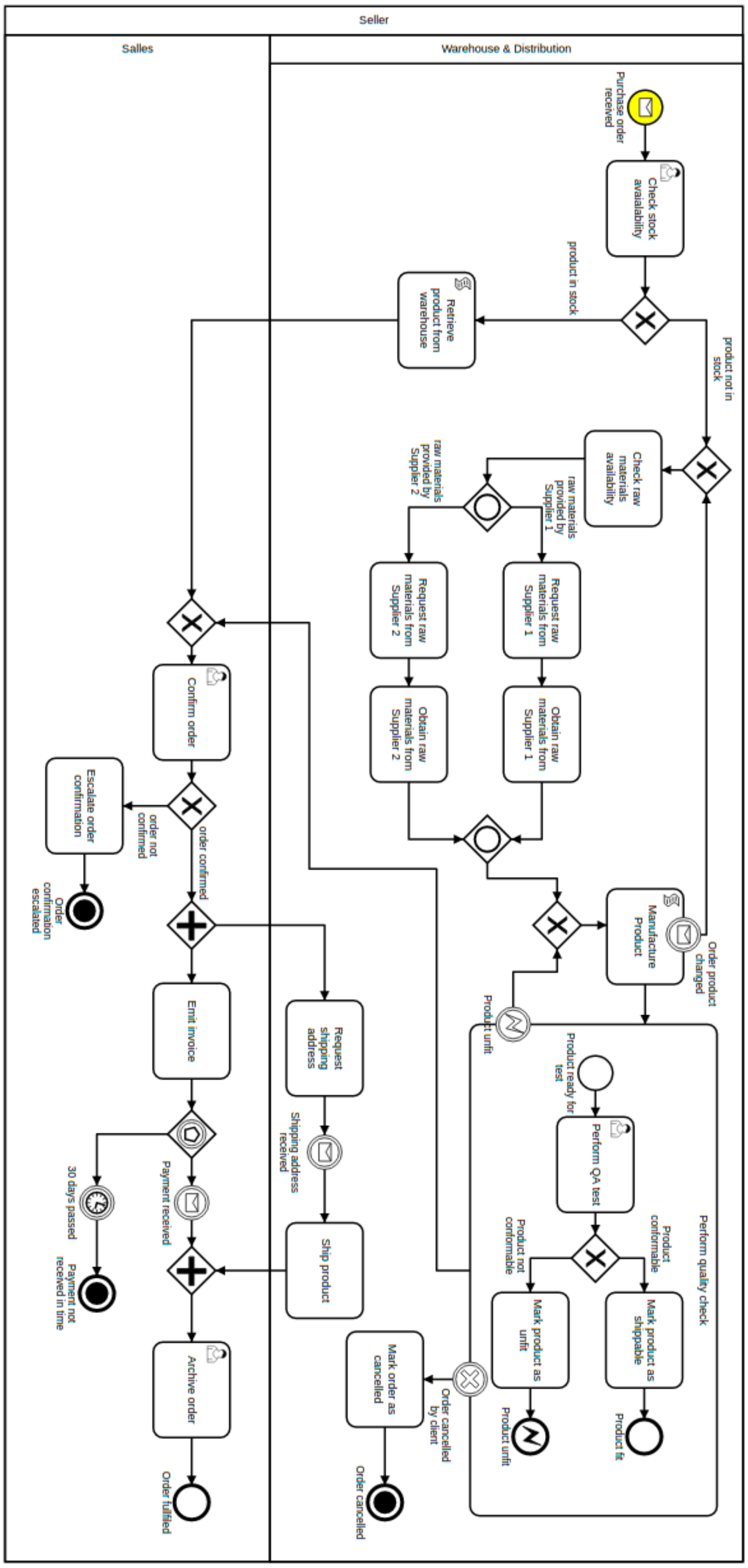


Figure 39: Model of the Case Study Process

The case study is presented in the form of a video presentation, with the purpose of presenting to the viewer the steps performed during the two phases of the process execution. The video can be seen at the web address provided in the references section in [32]. During the video presentation, the viewer can see how first, the business data is defined, and afterwards how the process is executed step-by-step, while manipulating and interacting with the data attached to the process.

5 Conclusions and Future Work

This final chapter is intended for introducing some ideas for potential future development of the tool and for presenting the final conclusions regarding the implementation. In addition, along with the conclusion some personal opinions and feeling related with the development of the system will be presented.

5.1 Conclusions

The implemented tool represents an original approach in the world of BPM process animations, due to the combination of its functional features. The implementation successfully combines the functionality to execute a BPMN process instance, with the ability to use visual forms in order to define and manipulate the business data attached to the process, and the option to execute the process instance in a distributed environment, by allowing multiple users to interact with the same process animation. In addition, the implementation represents a lightweight and self-contained tool, which allows the user to use it, without the necessity to install it or configure it in any way. The system does not have any external dependencies, such that, no third party software is required in order to be used. Due to the fact that the application is run in a browser window, we can say that the tool is cross-platform, since it can be used in the same way from any platform that supports a browser.

To conclude with, we consider that the developed tool bring an important contribution in the field of BPM process animations and that the functionality it provides can help the business analysts in their activities of process discovery and process redesign. The tool allow its users to interact and manipulate both the process execution flow and the business data attached to it, providing a realistic type of process animation. In addition, the application supports most of the core elements of the BPMN standard, therefore providing a useful tool, which can be used in order to simulate complex real-life business processes, in a straightforward and easy to understand manner, by providing clear information to the user as to the different states in which an element can be found during the process animation.

5.2 Future Work

Due to the fact that the field of work is of general interest to the BPM stakeholders, the future development of the tool can be done in many directions. However, in this section, only a few ideas will be presented, which in our opinion would represent the most important additions to the already developed system.

We can start by pointing out that the most important addition to the simulator would be the support for collaborations, represented by different pools and lanes. This would require also the support of message flows, in order to assure the communication mechanism between the different organizational entities of the process. However, the support of collaborations does not represent only the implementation of the BPMN behaviour of some new elements. In addition to that, the tool would allow the user to define roles during the animation, for each participant, such that one would only be able to execute elements which correspond to its own role. For example, if one user has the role of a logistic distribution department employee in a company, he should not be able to execute activities related to the financial departement of the company. In this sense, before starting the simulation, when the business data is defined, the roles of the users can also be created by the initiator of the process. Upon connecting to the process instance, each user would have to provide some credentials which would be used in order to identify the role that the he has during the execution of the process. By implementing this, the animations would benefit from a higher degree of realism, due to the fact that the process executions would be more close to the real-life situation, by allowing the tool to distinguish between different types of users.

Another future development idea, which would benefit the implementation a lot is the ability to export the diagrams in the bpmn format, meaning that the element data definitions would be defined inside each element, in the way specified in the BPMN standard. The benefit of this feature would be that the process model, along with the business data could be exported in a format which can be used by other tools, that implement the BPMN standard. At the moment the process, along with the business data is exported using a JSON format, therefore the saved process diagrams can only be used by the presented simulator.

From a BPMN semantics perspective, one enrichment idea is to support collapsed sub-process elements in the animation of the process. In the current implementation, sub-processes are supported only in the expanded form, which is not particularly helpful in reducing and encapsulating the complexity of the parent process. The expanded sub-processes are useful due to the fact that special types of events can be attached to a sub-process, such as cancellation or error events. However, if the user would be able to define in the process model a collapsed sub-process, which would be implemented in a separate diagram, the readability of the process model would be improved, while still providing the option for the user to attach special types of events to a sub-process. In this scenario, an option to reduce the complexity of a process would be provided, by encapsulating parts of it into different diagrams which can be modelled separately.

In the end, the last future functionality expansion idea is to implement some advanced BPMN element, such as non-blocking boundary events, multi-instance activity, or the event sub-process in order to support advanced BPMN features during the process animation.

6 References

[1] Jeston, John, and Johan Nelis. *Business process management*. Routledge, 2014.
[2] Dumas, Marlon et al. *Fundamentals of business process management*. Heidelberg: Springer, 2013.
[3] Weske, Mathias. *Business process management: concepts, languages, architectures*. Springer Science & Business Media, 2012.
[4] Silingas. “Business Process Modelling with BPMN”, 2012
[5] Allweyer, Thomas. *BPMN 2.0: introduction to the standard for business process modeling*. BoD–Books on Demand, 2010.
[6] White, Stephen A. "Introduction to BPMN." *IBM Cooperation* 2.0 (2004): 0.
[7] Owen, Martin, and Jog Raj. "BPMN and business process management: Introduction to the new business process modeling standard." (2003).
[8] Androcec, Darko. "Simulating bpmn models with prolog." *CECIIS-2010* 13 Jul. 2010.
[9] Freitas, Pereira. Process Simulation Support in BPM Tools: The Case of BPMN. Proceedings of 2100 Projects Association Joint Conferences, 2015
[10] Earls, Alan. "The rise and rise of BPMN for process modeling." 2011.
[11] Geiger, Harrer, Lenhard, Wirtz. “On the Evolution of BPMN 2.0 Support and Implementation”, University of Bamberg
[12] BPMN Introduction and History http://www.trisotech.com/articles/bpmn-introduction-history (last accessed on the 26th of March, 2016)
[13] E2ebridge - a tool which describes process oriented business logic https://github.com/e2ebridge/bpmn (last accessed on the 16th of May, 2016)
[14] Camuda - libraries for parsing, executing, and rendering BPMN 2.0 with JavaScript https://github.com/camunda/camunda-bpmn.js/ (last accessed on the 16th of May, 2016)
[15] Omi-Workflow - a workflow engine for the web http://workflow.omni-builder.com/quick-tour/ (last accessed on the 16th of May, 2016)
[16] Prime - process model animation tool http://prime.cs.vu.nl/prime.html (last accessed on the 17th of May, 2016)
[17] JSON - a lightweigth data-interchange format http://www.json.org/ (last accessed on the 15th of May, 2016)
[18] Dumas, Marlon et al. "Semantics of standard process models with OR-joins." *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS* (2007): 41-58.
[19] Völzer, Hagen. "A new semantics for the inclusive converging gateway in safe processes." *Business Process Management* (2010): 294-309.
[20] JavaScript - a lightweigth, interpreted, programming language https://developer.mozilla.org/en-US/docs/Web/JavaScript (last accessed on the 10th of May, 2016)
[21] Node.js - a JavaScript runtime https://nodejs.org/en/ (last accessed on the 19th of May, 2016)
[22] bpmn-js - a BPMN 2.0 diagram rendering toolkit https://github.com/bpmn-io/bpmn-js (last accessed on the 12th of May, 2016)
[23] JQuery - user interface oriented feature-rich JavaScript library https://jquery.com/ (last accessed on the 4th of May, 2016)
[24] Redux - a predictable state container for JavaScript applications http://redux.js.org/ (last accessed on the 15th of May, 2016)
[25] socket.io - a real-time application framework https://github.com/socketio/socket.io (last accessed on the 15th of May, 2016)

[26] socket.io-client - a real-time application framework https://github.com/socketio/socket.io-client (last accessed on the 15th of May, 2016)
[27] Scott W. Ambler. "Introduction to UML 2 Sequence Diagrams" http://www.agilemodeling.com/artifacts/sequenceDiagram.htm (last accessed on the 20th of Aprill, 2016)
[28] json-editor.js - JSON schema based editor https://github.com/jdorn/json-editor (last accessed on the 2nd of May, 2016)
[29] backbone-forms - form framework with nested forms https://github.com/powmedia/backbone-forms (last accessed on the 8th of May, 2016)
[30] bpmn.io - rendering toolkit and web modeler https://bpmn.io/ (last accessed on the 16th of May, 2016)
[31] Chrome browser https://www.google.com/chrome/ (last accessed on the 10th of May, 2016)
[32] Video Presentation of the Case Study https://vimeo.com/167318066 (last accessed on the 19th of May, 2016)

Appendix

I. License