Bachelor’s Thesis

Execution and Re-evaluation of BPMN Processes

Philipp Maschke

Supervision
Prof. Dr. Mathias Weske, Hasso-Plattner-Institute, Potsdam, Germany
M.Sc. Gero Decker, Hasso-Plattner-Institute, Potsdam, Germany
M.Sc. Harald Schubert, SAP AG, Walldorf, Germany

June 30, 2008
Abstract

Current workflow management systems do not properly support flexible business processes that require ad-hoc changes at runtime. This bachelor’s theses bases on a project that identified use cases and concepts for more flexible workflow systems in cooperation with SAP AG. Prototypical implementations were used to demonstrate the found solutions.

This thesis concentrates on two different areas of the project’s work. The first part of this thesis is centered around the issue of converting and configuring BPMN models in such a way that they can be executed in a flexible manner. In the second part, the feature of re-evaluation is dealt with, which covers the issue of handling external changes on data objects of processes while these are being executed. The issue is motivated by a real-world example, the conceptual solution is explained and details to the prototypical implementation are given.

Zusammenfassung

Derzeitige Workflowmanagementsysteme unterstützen flexible Geschäftsprozesse, die ad-hoc Änderungen zur Laufzeit verlangen, nicht zufriedenstellend. Diese Bachelorarbeit basiert auf einem Projekt, das in Zusammenarbeit mit der SAP AG Anwendungsfälle und Konzepte für flexiblere Workflowsysteme identifiziert hat. Prototypische Implementationen wurden dabei genutzt um die gefunden Lösungen zu veranschaulichen.

# Contents

1 Introduction  1

2 Bringing BPMN to execution  3
   2.1 Example Scenario - Hire-an-Employee  3
   2.2 Converting BPMN to executable Petri nets  4
      2.2.1 BPMN Task Lifecycle  5
      2.2.2 The Context Place  11
      2.2.3 Ad-Hoc Subprocesses  12
      2.2.4 Data Objects  13
   2.3 Architecture Overview  14
   2.4 Implementation  16
      2.4.1 Oryx’ BPMN to Petri net converter  16
      2.4.2 Changes on Converter  17
      2.4.3 Petrinet Execution Engine  19
      2.4.4 Changes on Execution Engine  21

3 Re-evaluation of BPMN Processes  25
   3.1 Leading Scenario - SRM Approval  26
   3.2 Conceptual Model  27
   3.3 Modeling Implications  31
   3.4 Prototypical Implementation  33
      3.4.1 Transition Modes  33
      3.4.2 Execution-Log and -Context  36
      3.4.3 Enhanced Firing Semantics  37
   3.5 Outlook  39
4 Conclusion 41

Bibliography 43
1. Introduction

This bachelor’s thesis is part of the documentation for the bachelor’s project “Ad-Hoc Business Processes” carried out in the academic year 2007/08 at the “Business Process Technologies” chair of the Hasso-Plattner Institute in Potsdam.

Business process models are a great tool for defining, illustrating and realizing how businesses actually work. Thus, business process management systems have gained a lot of influence in recent years. By using process models as input for a generic workflow engine, they also simplify the development and execution of business processes. The existing solutions [12] work especially well with highly structured and repetitive processes.

On the other hand, current systems require the processes to be modeled inside a very rigid structure, which often cannot be done for certain types of processes. This is especially true for creative processes that are executed manually. There, human workers contribute a lot of individual knowledge and experience, thus shaping the process as it is best in the current situation. Hence, different instances of these processes usually differ not only in data, but also in their structure and number of participants.

Because traditional workflow management systems provide neither the modeling tools nor the execution concepts for flexible processes, they cannot support these processes properly.

Bachelor’s Project

The bachelor’s project, that this thesis is based on, had the goal of examining the degrees of ad-hocness that can be supported by workflow management systems. Together with SAP AG and the team at the chair of Prof. Weske, use cases were analyzed, concepts created and prototypes developed in order to support more flexible processes.
The SAP AG is currently developing a new workflow management suite based on the Business Process Modeling Notation (BPMN [1]). The product *NetWeaver Business Process Management* (code name: Galaxy) aims to cover traditional workflows as well as ones with more flexible behavior. Therefore, the identification of relevant concepts in the latter area was an important issue for this project. Besides Galaxy, the Oryx platform was used as a second workflow managment system. This platform is a scientific open-source solution that can be extended more easily, amongst others because of its smaller size. It is based on a modeling tool, developed in a previous bachelor’s project [11]. Here also, BPMN is one of the supported modeling notations.

In the beginning of the project, we examined related work in the area of flexible workflows and collected business and also real-world use cases, which should provide more detailed requirements for the prototypes that were to be created. For this purpose, we also visited the SAP Galaxy developers and potential users in Walldorf. Besides close contact with Walldorf during the whole project, SAP researchers from Karlsruhe and Brisbane also participated in meetings and calls.

During the project, we identified three major concepts: design-time based adhocness, delegation and re-evaluation. Prototypes supporting the identified concepts were implemented, both within Galaxy and Oryx. Because the runtime structure of Galaxy, the Process Server, was too complex to be comprehended and extended using only the limited resources of our project, we limited the efforts on the Galaxy side to the modeling environment (Process Composer). To prove the concepts that were developed, prototypes were implemented in Oryx.

Besides this thesis, five others have been written in association with this bachelor’s project: The two theses [8] and [6] handle the topic of design-time-based adhocness and an overview on identified use cases and the resulting usability considerations is given in [7]. [10] shows the concepts and implementations in the area of delegation, while in [5], changes on Oryx are specified and concepts for process analyzation are explained.

This Bachelor’s Thesis

This thesis is organized as follows: Chapter 2 explains, how the BPMN models were converted and configured to make them executable in a flexible way. It does so by illustrating the concepts for the conversion first and afterwards showing how it was implemented in the Oryx environment.

In chapter 3, the concept of re-evaluation is covered. For this purpose, a motivating example is illustrated, which we obtained from the SAP AG. Afterwards the conceptual solution for this problem is explained and finally details to the prototypical implementation of this solution in the Oryx environment are given.

Chapter 4 gives a conclusion by summing up the work done for this thesis as well reflecting on the project as a whole.
2. Bringing BPMN to execution

The aim of our bachelor’s project was to take the Business Process Modeling Notation[1] as a basis to explore and evaluate different ad-hoc constructs. In this context, ad-hoc means “for a specific purpose or situation and no other”. Thus, it implies taking action during the execution of one specific instance of a process, without affecting other instances of that process. Of course, to be able to evaluate different ad-hoc constructs, we first needed a platform on which we could bring BPMN models to execution. Since BPMN in its original form does not contain all information necessary to execute its models directly, we had to translate it into a language, which could then be executed. For this purpose we used a concept very similar to the well-known concept of Coloured Petri nets. For further details on this concept see Section 2.4.3. This section covers the topic of how we translate BPMN models to Petri nets.

2.1 Example Scenario - Hire-an-Employee

In this section I will introduce one of the scenarios that had to be supported by our prototype. The concepts and implementation details of the following sections will be explained with the help of this scenario:

A company wants to hire a new employee. They already did an assessment center and picked one applicant to be hired. Now a multitude of tasks have to be done:

- his social insurance information and other data have to be collected
- a workplace has to be set up
- he may need to get a security card
- his profile has to be added to the company’s homepage
These tasks can be done in parallel and in almost any order. Almost, because some tasks depend on data that other tasks create. The workplace, for example, can only be set up when the new employee’s job type and room number are known. The model for this scenario can be seen in Figure 2.1. We see that the task ‘Issue Security Card’ has a question mark in the upper right corner. This represents the fact, that it is not needed every time and thus can be skipped. Other important constructs used there and those deviating from the BPMN specification[1] are partly explained in Section 2.2.3. A more detailed description is included in the bachelor’s thesis [6].

Figure 2.1: Hire an employee model

This scenario has an additional twist to it: Sometimes, not all information needed for the new employee’s form is included in the original application form. When this is the case, the secretary fills in all information that is provided and then tells the new employee to fill in the rest himself and also check the information for correctness. After she delegated the task and the employee filled in the missing information, she reviews the completed form and then submits it into the system.

2.2 Converting BPMN to executable Petri nets

The following section deals with the conceptual translation of certain BPMN constructs into Petri nets.

There is, of course, already a basic mapping for translating BPMN to Petri nets. This mapping is fairly straight-forward and creates Petri nets that have semantics
that are similar to the original BPMN models. It includes constructs for exception handling and basic BPMN elements, whose translation can be seen in Figure 2.2.

<table>
<thead>
<tr>
<th>BPMN Object</th>
<th>Petri-net Module</th>
<th>BPMN Object</th>
<th>Petri-net Module</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Start s" /></td>
<td><img src="image" alt="s" /></td>
<td><img src="image" alt="End e" /></td>
<td><img src="image" alt="e" /></td>
</tr>
<tr>
<td><img src="image" alt="Message E" /></td>
<td><img src="image" alt="E1" /></td>
<td><img src="image" alt="Task T" /></td>
<td><img src="image" alt="T1" /></td>
</tr>
<tr>
<td><img src="image" alt="Fork F1" /></td>
<td><img src="image" alt="F1" /></td>
<td><img src="image" alt="Join J1" /></td>
<td><img src="image" alt="J1" /></td>
</tr>
<tr>
<td><img src="image" alt="Decision D1" /></td>
<td><img src="image" alt="D1" /></td>
<td><img src="image" alt="Merge M1" /></td>
<td><img src="image" alt="M1" /></td>
</tr>
<tr>
<td><img src="image" alt="Receive task T1" /></td>
<td><img src="image" alt="T1" /></td>
<td><img src="image" alt="E1" /></td>
<td><img src="image" alt="T1" /></td>
</tr>
</tbody>
</table>

Figure 2.2: Mapping task, events, and gateways onto Petri-net modules

For further information on the standard mapping see [4]. For the purpose of our project, we needed a mapping that supports some additional features, though.

### 2.2.1 BPMN Task Lifecycle

**Requirements & First Extension**

The standard mapping described in the previous section creates a Petri net with similar semantics to the original BPMN model. However, it does not create a Petri net, whose behaviour conforms to a BPMN model when executed. The first reason for this, is the fact that a BPMN task can have several states, like **initialized**, **enabled**, **running** or **completed**. Secondly, the firing of a task can take a considerable amount of time whereas the firing of a transition conceptually takes no time. Also, in Petri nets there is no concept of users or roles. But since these concepts are a vital part of any business process management suite, we needed to teach Petri nets how to cope with that.

So, in order to get a Petri net that behaves like an instance of a BPMN model, the mapping algorithm for a task had to be extended. In the following, the collection of
transitions and places representing one task will be called a task subnet. The first extension was to split the firing of a task into a start and end transition with an intermediate place. That way the second problem mentioned above, of a transition firing in virtually no time, is solved.

BPMN task states

As mentioned above, a task can have several states during the execution of a process. The states as stated in the specification[1] are amongst others:

- initial(none): task exists, but cannot be executed yet
- ready: task is ready to be executed
- running(active): task is being executed
- completed: task has been executed

The listing above leaves out some states, because they did not fit into the concept of Petri nets. For instance, 'completing' can have no representation in a Petri net, because changes of state happen in virtually no time.

We soon discovered that these states are not sufficient to meet the needs of the scenario described in Section 2.1. When we look at the task 'Issue Security Card', we need to add the possibility to bypass its execution. We do this by introducing the state skipped, which can be reached from the ready state and belongs to the parent state 'done'. Note that this state is optional, meaning that it will only be available when the task is marked as skippable.

We also want users to be able to suspend a running task, thereby pausing its execution and saving the current results without submitting them. The user can then execute some other tasks or go home and resume working on the suspended task the next day.

The scenario described in the previous section also calls for the ability to delegate a task to someone else and afterwards review that task if wanted. To be able to realize this feature we need two more states for a task, which are 'delegated' and 'under review'. An overview of all states and the possible shifts between them are shown in Figure 2.4.
2.2. Converting BPMN to executable Petri nets

Roles & Users

As mentioned in the beginning of this section, these concepts are usually not considered in Petri nets. Transitions typically just fire and there is no role or user connected to it. So before we could implement the lifecycle and later on the delegation feature, we first had to realize that a task subnet can have one specific owner. The user that owns a task subnet must be able to control its execution, without anyone else being able to interfere.

To do so, we firstly extended transitions in a way, that an execution role can be specified and only users having that role can fire them. Secondly, the actual user executing a task subnet is stored as its owner in the corresponding context place (see section 2.2.2). As long as a subnet has an owner, only he can fire its transitions and nobody else.

Simple Lifecycle

With that done, we could extend the translation of a BPMN task into a corresponding Petri net construct, so that it complies with the states we defined above.

Figure 2.5 shows the changes on the resulting Petri net. The constructs for delegation and suspension are left out for the time being, since they cause the resulting net to be a little bit more complicated.

The first transition in a task subnet is 'tr_enable'. It fires automatically when a valid combination of input tokens exists and puts one token on the place 'pl_ready'. Its firing represents the change in state of it's task from initial to ready. Afterwards the task can be allocated. This means that a user, who's role complies to the role of...
2. Bringing BPMN to execution

The task, reserves it for himself. When fired, the token from ‘pl_ready’ is moved to ‘pl_running’. If the corresponding task was marked as skippable, then the ‘tr_skip’ transition is added to the task subnet. It can be fired instead of ‘tr_allocate’ and moves the token from place ‘pl_ready’ to ‘pl_complete’. This causes the transition ‘tr_finish’ to fire automatically, putting one token on each outgoing place. If the task was allocated, then it can be executed and the results can be submitted by firing ‘tr_submit’. This moves the token from ‘pl_running’ to ‘complete’, which also causes ‘tr_finish’ to fire. There is no explicit place that can be used to detect if a task has not been executed yet, if it was completed regularly or has been skipped. The solution for this problem is described later in this section.

Suspend & Delegate

So now, we would like the task to be suspendable. We achieve this by introducing the two new transitions ‘tr_suspend’ and ‘tr_resume’ with the intermediate place ‘pl_suspended’. Tokens lying on this place cache the data of the suspended task, that were filled in until the suspension (see section 2.4.3). The result of translating a task with these extensions into a Petri net module can be seen in figure 2.6.
Now, we want to be able to delegate a task to a specific person. To do so, we need to set the current owner of the task to that person. One thing to consider here is that this only works, when a task is delegated to a person with the same role. In the scenario though, the secretary needs to delegate her task to the new employee, who definitively does not have the role 'secretary'. The solution we brought up was to enable the current owner of a task to fire each transition of this task without checking for his roles. So when 'tr_alloate' is fired, the user firing it is checked for a complying role and then - if the check was successful - set as the current owner of the task. Afterwards no role checking is done anymore.

The task subnet is extended as follows: To change the owner stored in the subnet and specify some other options, we added the transition 'tr_delegate'. Because of technical restrictions it was placed, so that it can fire only after 'tr_submit' has been fired. As it turned out, these restrictions could have been averted, but up to the time this thesis was written, the needed changes have not been implemented. To achieve, that only 'tr_delegate' can fire when a user wants to delegate, a 'delegated' flag is set to true. This flag is then checked in the guard expressions of the following transitions. These guard expressions can be specified for each transition and check whether it can fire or not. More on this in Section 2.4.3.

The 'tr_delegate' transition sets the specified delegate as the new owner, when the transition fires. This causes the task to be already enabled and allocated to the delegate. He can now work on the task and afterwards submit his results.

Then, there can be two different transitions enabled. Either 'tr_review', if the original owner wanted to check the results of the delegate, or 'tr_done', if he didn't want to check. This choice also is realized through guard conditions and a special flag.

The resulting Petri net created by all the extension previously described can be seen in Figure 2.7. Further concepts like multiple delegation and their implementation

![Figure 2.7: Task lifecycle with delegation](image-url)
details are discussed in [10].

**One marking but several states**

One of the goals of the conversion process was that for each state of a BPMN task, the corresponding task subnet would have one designated state. The state of a Petri net is represented by the distribution of tokens on places, the so-called *marking*.

![Diagram of task subnet markings and corresponding task states](image)

We can see in figure 2.8 that a task subnet can have markings that represent more than one task state and that one state can be represented by two markings. The places causing this particular situation are `pl_running`, `pl_deciding` and `pl_complete`. Now, a number of condition variables, which are stored within the tokens, make the difference between the corresponding task states. For example: When we are in marking M4, then the condition 'delegated' differentiates between the states 'running' and 'delegated'. Likewise, when in marking M5, the condition 'isReviewed' decides between 'running' and 'under_review'. And finally, a status variable distinguishes between 'skipped' and 'completed' when being in marking M6. This variable is also used to distinguish between 'initial' and the 'done' states after the task is finished. This differentiation is needed, because the subnet then goes back to it’s initial marking M1.
2.2. Converting BPMN to executable Petri nets

To express the differences of the states inside the subnet structure, we could have added places to represent this. For example, a place ‘delegated’ could have been added, which would have been filled by the transition ‘delegate’ and cleared by a new transition that would be enabled by ‘submit’. But this would have made the subnet for a single task much more complicated and confusing. Instead we chose to add one designated place, which we called the context place, that should contain the condition variables mentioned above, additional information about the tasks current state and other meta-data.

2.2.2 The Context Place

As introduced in the previous section, the context place is a special construct for keeping token, which contain meta-data about a certain task. This data could include the current, detailed state, the current owner, special flags needed for delegation, start time, end time and much more.

Each task gets a context place in its Petri net representation and there is one token on it per each case of the corresponding process. Each transition within the task subnet reads and writes onto the context everytime it fires. That way the previous data is not lost and still new data can be added.

![Diagram of a context place with connections to task subnet partially shown.](image)

Figure 2.9: Context place with connections to task subnet partially shown

The purpose of this place is to concentrate all meta-information of one task instance into one token, thereby creating a single point of access for outside and inside constructs to get hold of these information. The centralized data is used by guard conditions within the task subnet and also by guard conditions checking the completion condition of ad-hoc sub-processes. It is also used to store and access execution information, which is displayed to the user or an administrator. Also, it is used to log actions and changes of state of tasks, which user caused them and the time they happened.
Another main reason for the realisation of this construct, is the clean separation between process data and meta data it creates. In the beginning of the implementation phase all data was stored within the one token (or multiple) that roamed around the net. That brought up problems concerning where to get current information from. Also we saw problems, when there were parallel paths. At the split transition the data was duplicated and during the parallel paths concurring changes could take place, which then would have to be merged together.

After some discussion we decided to go for a realization of this construct, although in the end it proved to be a major effort to do so. The implementation and it’s problems are described in Section 2.4.

2.2.3 Ad-Hoc Subprocesses

These special sub-processes are a BPMN construct. The specification [1] defines them as follows:

"The activities within an Ad Hoc Embedded Sub-Process are not controlled or sequenced in a particular order, (their) performance is determined by the performers of the activities."

In other words, an ad-hoc subprocess contains a number of activities or subprocesses, which can be executed in any possible order. The example scenario (section 2.1), already introduced a situation where such a construct is needed.

There are some attributes for ad-hoc subprocesses to specify their behaviour. Firstly, one can set an Ad-Hoc Ordering option to two different values. The first is 'parallel' and expresses, that several tasks can be executed at the same time. This is the default value. The other is 'sequential' and limits the execution of the tasks to a maximum of one at a time. This can be useful when there are restrictions on resources or performers.

The second attribute is the so called Completion Condition, which is an expression for specifying when the subprocess is in such a state that it can finish. The BPMN specification gives no more explanation as to how powerful this expression can be. Can it use information on current task states, included or outside data objects or even context data like time? Even the syntax of this expression is left undefined.

That’s probably one of the reasons, why this construct has never been implemented in a publicly known tool until now. Another critical point that’s unclear from the specification is, whether included tasks can be executed multiple times or at most once. For the purpose of our implementation, we clear this matter by specifying that tasks can be executed at most once.

In our project work we developed Petri net constructs, that enable us to execute ad-hoc subprocesses. We support both the sequential and the parallel case and also
the modeling and evaluation of completion conditions. These conditions can be specified, using task states as well as included data objects’ attributes. An example of an BPMN ad-hoc subprocess modeled with Oryx is shown in Figure 2.10. The annotations are just for illustration purposes, because the actual attributes are not visualised in the model. The actual translation of ad-hoc subprocesses and a more detailed view can be found in [8].

2.2.4  Data Objects

Data objects are a topic that is not covered consequently in the BPMN specification, although it is included. There is no concept of how the data is or can be structured, about lists, enumerations or other kinds of containers, about the lifecycle of data objects, their standard values, concurrent access, versioning and all the other important topics.

Because of this rather stepmotherly covering of data objects, they are not included in the basic mapping from BPMN to Petri nets [4].

Of course, for prototypes that are supposed to cover the execution of ad-hoc BPMN processes and re-evaluation, we needed a lightweight support for data objects. Not very surprisingly, we translate each data object into one data place. We extended the BPMN model, such that an xml schema can be defined for a data object. This schema is then taken as a template for the tokens lying on the corresponding data place.

We also assume that we can use read arcs for read-only access on places in underlying execution engines to solve the problem of concurrent access to data. As long as it’s only read access we have no problem. Each task gets it’s own local copy of a data object when it starts being executed. If a task writes an object it does so only when it finishes, where it removes the old data token and replaces it with the new one.
This is illustrated in Figure 2.11. Currently there is no versioning, merging changes or similar concepts used in our solution, but it is open for extension.

The user defined xml schema of the data objects is also used to automatically generate the user interfaces for tasks, which minimizes the work for a user of our prototype when deploying a net. For details on this concept, more on the implementation of data objects or user interfaces see [7].

### 2.3 Architecture Overview

This section gives a short overview on the prototype that we developed at the Hasso-Plattner Institute for this bachelor’s project. It is spread out over four parts, which mostly existed before, but partially had to be greatly extended and connected with each other.

Firstly, there is the Oryx modeling environment, which was developed in the previous bachelor’s project at the ‘Business Process Technology’ chair. It can be used to create
models in a collaborative way on the web without having to install any software other than an internet browser. A very flexible plugin system and the definition of model languages through stencil-sets enables easy extension of interfaces and functionality for Oryx. The bachelors thesis of Willi Tscheschner[11] provides a more detailed look into the Oryx modeler.

BPMN models created with Oryx can then be converted into a Petri net model through the BPMN2PN converter. The framework for this converter already existed, but had to be extended to supply the executable form of Petri nets described in Section 2.2. The input for the converter are BPMN models that are specified in the RDF interchange format. The converted Petri net conforms to the PNML format, a standard for Petri net interchange specified in [2].

The converted Petri nets are then deployed onto a Petri net engine, that was developed as a seminar work by Kai Schlichting and Alexander Lüders[9]. It is a resource-oriented, web-based execution engine for Colored Petri Nets. Embedded inside a webserver, it communicates via HTTP with users, external tools and with other execution engines as well.

The nets deployed onto the engine can then be manipulated via the worklist. This is an interface for the end-user where he can view his tasks, execute them and retrieve information about running processes. This part was the only one that did not exist before and so had to be created during our project. It was designed as an attachment to the Petri net engine.

A detailed view on the four parts and their integration to one prototype is covered in [5].

\[1\text{http://www.w3.org/RDF/}\]
2.4 Implementation

After I explained the concepts of what we did, to convert BPMN to Petri nets in Section 2.2, I will now take a closer look into how we did this.

To implement all the concepts we wanted to support, we had to extend Oryx’ BPMN to Petri net converter as well as the Petri net execution engine introduced in the previous section.

2.4.1 Oryx’ BPMN to Petri net converter

The converter is currently realized in two parts, a plugin for Oryx and a Java servlet for the actual conversion. The plugin adds two buttons to the interface of the modeler, one for triggering the standard mapping and one for our extended conversion algorithm. Once clicked, the plugin extracts the current version of the model out of the browsers DOM, converts it into the RDF format and then calls the conversion servlet with the RDF as argument.

![Figure 2.14: Conversion process as BPMN model](image)

The converter then parses this structure and builds up an internal BPMN metamodel. That model is then used for the actual conversion into a Petri net metamodel. This is done by a ‘Converter’ class, which provides handle methods to convert the different BPMN elements. This class can then be specialized for each kind of Petri net, like it was already done for standard Petri nets (‘StandardConverter’) and executable Petri nets (‘ExecConverter’). However, in the beginning these two classes did not contain or overwrite any handle methods.

Afterwards that metamodel is converted into the PNML format, which is then automatically exported to a specific engine, that can be specified inside the converter. This automatic deployment is skipped, when the standard conversion algorithm is used. This process is illustrated as a BPMN model in Figure 2.14
The internal metamodels are made up of elaborate class hierarchies, which on one hand enable easy extension and good reuse of code. But on the other hand they are quite complex and it needs some time to get acquainted with the restrictions and dependencies between all the interfaces, factories, inheriting and implementing classes. Also we had some problems and lengthy discussions about how and where to extend, inherit or create new classes and interfaces, such that it all fits nicely into the given framework. Sometimes we just chose to break the restrictions the hierarchie imposed on us, to be able to quickly implement and test extensions.

2.4.2 Changes on Converter

Following the path of conversions described in the previous section, the main points of interest for us where the BPMNRDFImporter, the ExecConverter and the ExecPNLEExporter. The RDF importer had to be made aware of changes to the BPMN models, which were necessary from time to time. For instance, the attributes 'AdHocOrdering' and 'CompletionCondition' for ad-hoc subprocesses were missing in the original stencil-set. Also, attributes for an xml-schema for data objects or roles for tasks had to be added. These additional attributes had to be added to both internal metamodels and propagated by the RDF importer as well as the PNML exporter.

Apart from these three classes there were some other points that we extended or added. For example, the support for data objects was implemented in the 'Converter' class, since this is a concept that is needed in all conversions of BPMN models to Petri nets. Of course we also added and modified several other classes like 'TransformationTransition', 'FormTransition' and so on, but the hubs of the conversion algorithm are the three mentioned classes.

ExecConverter

This class implements the concepts task lifecycle and ad-hoc subprocesses. Therefore it specializes the two methods 'handleTask' and 'handleSubprocess'. To realize the task lifecycle we created a specialized BPMN task class, the 'ExecTask'. In this class we specify variables for all places and transitions that belong to a task subnet. When 'handleTask' is called we get an instance of the standard task class, whose attributes we use to instantiate an instance of 'ExecTask'.

Afterwards all places and transitions are created one by one, their attributes set and connections between them created. One thing to mention here, are the different transition types we used. We have several different ones, all behaving differently concerning user interaction. We need these, in order to make the user to believe that he is executing a BPMN task and not a collection of transitions. All transition types have an id, a label and know their corresponding BPMN task's designator.

AutomaticTransition Firstly, there is the AutomaticTransition. It fires automatically as soon as there is a valid input token combination and uses an XSL stylesheet to transform incoming token values. When none is given, it uses a default
transformation to merge all input tokens into one model and copy that model onto all outgoing places. The need for this special *copy-xslt* arose, because the standard transformation in the engine, which is used when no other is given, can only copy one incoming token. This makes it useless as soon as there are more than one incoming connections. The enable, complete and finish transitions are of this type.

On thing to note here, is that the engine *only supports one output model* per firing. To achieve that different outgoing places get different token values, one has to add arc transformations in the form of XSLTs to the corresponding arc.

**TransformationTransition** The transitions allocate, skip, suspend and resume are of the type *TransformationTransition*, which also has an XSL stylesheet to transform incoming tokens. The difference to the automatic transition is, that the transformation transition is fired through user interaction. Also, transformation transitions have an attribute ‘action’, which is used by the worklist to determine, which transition is responsible for which change in state for the task. This attribute is not needed for automatic transitions, because they don’t require user interaction and therefore don’t need to be handled by the worklist. So this kind of transition is for situations, where something has to be triggered manually, but the actual work should be done without user interaction.

**FormTransition** The third type of transition is the *FormTransition*, which is similar to the transformation transition, just that the automatic transformation in form of the XSLT is replaced by a manual transformation in form of an XForms document. Also, this transition needs an additional bindings document that connects corresponding input- to output-variables. This type could be called a truly manual transition, because it is triggered manually and the output is determined manually as well.

Also, for each data place and each context place, so called locators have to be defined. Locators actually are just an XPath expression that has to be defined for each data attribute, that one wants to check in a guard or access in any other way, without using XML transformations. The expression is combined with a name for the attribute. This name together with the identifier of the place can be used to reference the data that is returned by the XPath expression. This concept is used in the the Petri net execution engine, but already has to be defined in the conversion.

**ExecPNMLExporter**

This class is responsible for extracting all information out of the Petri net metamodel, that is needed to specify and export a PNML document that represents an executable Petri net. It does so by iterating over all places, transitions and arcs, gradually building up the wanted document.

PNML is a standardized interchange format for Petri nets, which just provides basic constructs for describing Petri net elements. Therefore not all information necessary
for us can be natively described in it. But for this purpose there is the idea of
the 'toolspecific' tag. Under this tag there can be any XML data, needed for a
specific tool, in our case the execution engine. Here we put most of the attributes
describes above, like the address where an XSL stylesheet, XForms document or
binding can be found, the action, the parent task, the type of transition and several
more. Because exporting these attributes includes many recurring activities, we also
created a helper class to simplify them.

2.4.3 Petrinet Execution Engine

The Petri net execution engine is the result of a seminar work[9] on process-oriented
information systems at 'Business Process Technology' chair. The goal was to develop
a RESTful process engine for Petri nets using PNML as the interchange format. Also
the Atom Publishing Protocol was to be used for realizing places as feeds and tokens
as their entries. This feature enables the engine to execute nets with a high level
of interaction together with other engines or external objects. One example for this
would be to use feeds on Ebay searches to trigger process instances. As said before,
the engine supports the notion of Coloured Petri nets, where each token has an
XML document as its value. This enables very powerful concepts, but also creates
problems concerning complexity and performance.

The engine’s architecture is fairly simple. Embedded within a webserver, the en-
gine handles HTTP-requests on its resources, triggering different handlers within
its model-view-controller framework. It uses its own as well as external libraries
to cover areas like PNML parsing, evaluating XPath expressions or processing XSL
stylesheets.

It currently supports three different kinds of transitions: automatic, receive and send
transitions. Of these transitions, only the first two are of interest for us, because the
third is needed only for interacting with other engines.

AutomaticTransition

This transition is triggered automatically as soon and as long as there are active
input-token combinations. When firing, it gets an XML document as input from the
engine core, containing the values of all tokens to be consumed by that firing. That
document is then transformed by an XSL stylesheet. This can either be specified
beforehand or if none was given, a default stylesheet is used, which copies only one
input token into the output. The resulting XML document is taken as output model
of this transition.

ReceiveTransition

This transition is triggered only by receiving the request to fire. This request can
come from a user, another engine or some external source. The request must contain
the identifiers of the tokens, that are to be consumed, and the output XML document. This document is then taken directly as the output model. Receive transitions usually have an XForms document attached to them to provide an interface for the user. That interface can be used to express, which data should be used to create the output model. If no XForms document is supplied, then a standard interface with just a submit button is used.

As hinted in the previous paragraphs, a transition only has one XML document as its output, even if there are multiple outgoing places. Thus, a token with one and the same value is put onto each outgoing place. The only way to achieve differing token values for different places, is to specify arc transformations on the corresponding arcs from transition to place. An arc transformation consists of one XSL stylesheet, which takes the transition’s output model as input and creates an XML document. This document is then used to fill the token, which is then put onto the corresponding place. So the firing of one transition can lead to the processing of several XSLT's. One for transforming the transitions input values into an output model and at most
one per outgoing arc, for transforming that model into an XML document for the
place lying behind the arc.

2.4.4 Changes on Execution Engine

Several changes had to be made to support the task lifecycle and the context place
concept in the given engine.

Firstly, we needed to be able to trigger transitions, but not worry about input
or output data. This behaviour is needed by allocate, suspend and some other
transitions of the lifecycle. To do so we had the choice of either adding an option
for automatic transitions to be triggered manually or adding the possibility to assign an
XSL stylesheet for receive transitions. After some discussions internally and with
the developers, we went for the second choice, because it was easier to implement.

The second concept, the context place, took some more time to implement. Because
each transition needed to update the context place, which includes reading the old
data and updating certain values, now they all had it as an incoming place. But since
transitions only have one output model, we had to find a construct for extracting
the context data out of the transition’s output model. Also the remaining data had
to be extracted, to be then put onto the next place in the control flow.

This separation was realized by developing special XSL sheets for each transition in
the task subnet. These ‘context-xslts’ are used as arc transformations on the arcs
from transition to context place and extract the metadata. Also, there is another
XSLT for extracting processdata out of the outgoing model, which is used on each
arc from transition to next place. As can be seen in the following code extract, the
context-xslt for the enable transition takes the initial context token and copies all
values, except the current status, which is set to ‘enabled’:

```xml
<xsl:template match="/"
    <data>
    <xsl:apply-templates
       select="//@metadata[contains(@place_id,’pl_context’)]"/>
    <!-- current assumption: only one context place exists -->
    </data>
</xsl:template>

<xsl:template match="metadata">
    <metadata>
    <xsl:apply-templates/>
    </metadata>
</xsl:template>

<xsl:template match="metadata/status">
    <status>enabled</status>
</xsl:template>

<xsl:template match="metadata/*">
    <xsl:copy-of select="."/>
</xsl:template>
```
The allocate transition’s transformation also sets a value for the current state, but additionally sets the start time, current owner and first owner fields:

```
<xsl:template match="/">
  ...
  <xsl:template match="metadata/startTime">
    <startTime><xsl:copy-of select="$time" /></startTime>
  </xsl:template>

  <xsl:template match="metadata/status">
    <status>running</status>
  </xsl:template>

  <xsl:template match="metadata/owner">
    <owner><xsl:value-of select="$user" /></owner>
  </xsl:template>

  <xsl:template match="metadata/firstOwner">
    <firstOwner><xsl:value-of select="$user" /></firstOwner>
  </xsl:template>

  ...
</xsl:template>
```

Before the development of the different arc transformations, we did not specify a specific XSLT for automatic transitions and implicitly used the 'cp_one_token.xsl' file. As you can see below, the template defined there does not match on elements of an XML file, but is rather called directly by the 'main.xslt.xsl', which is a framework XSL provided by the engine for each transition transformation. It collects all incoming tokens and stores all their data into the 'token.xml' parameter and then calls whichever template was defined under the name 'PNENGINE'.

```
<xsl:template name="PNENGINE">
  <xsl:param name="token.xml" />
  <xsl:for-each select="exsl:node-set($token.xml)//place">
    <xsl:copy-of select="document(.)" />
  </xsl:for-each>
</xsl:template>
```

Because of the additional context place we became aware of the fact that this XSLT was only useful with exactly one incoming place. The reason for this is, that it creates several root nodes, when there are multiple incoming tokens, but only one root node can be put onto following places. Now we mostly had two incoming places and all automatic transitions refused to work properly. So we had to develop an extended version of a copy-xslt, which worked with multiple incoming places and merged the different tokens under one root node.
Here we rely on the fact that all values of tokens are placed between a 'data' tag by the engine. This transformation file was subsequently extended several times to allow for copying runtime information from within the engine into the output model of a transition. This is used to get a user, who is currently logged in on the engine, to become the owner of a task he is allocating. The extended copy transformation also provides information about from which place certain xml tags came from. This is used by the different context arc transformations to recognize, which values came from the context place and therefore have to be extracted.

In figure 2.16 an example is given for the XML transformations taking place, when 'tr_allocate' fires. We can see the previous token values, the transition’s output model and the results of applying the different arc transformations on that output.
The implementation of the context place was seriously delayed by several factors. For one there were inexplicable errors thrown by the engine’s XSLT processor, which did not appear when an external processor was used. This error was bypassed after some time with the help of the developers. It did also take an considerable amount of time to find out just how, when and what kind of XML data was produced or transformed by the engine and how exactly one had to write the XSLTs to make them work. This was of course on the one hand caused by the fact that we did not have much experience on that matter, but on the other hand wasn’t eased by the fact that virtually no documentation on that topic was available in the engine.

Development on the engine was usually rather slow and fairly complicated, because it was and still is in a prototypical state. Several times we had the case that, after finally getting a feature to work, some change on the engine caused it to stop working again. We considered creating a branch for our development, to retain a stable environment for our development. But we discarded the idea, because the changes done by the developers were usually either fixing major bugs or adding functionality that we needed.
3. Re-evaluation of BPMN Processes

During the course of our project some of our initial goals where specified more clearly and some were dropped, because of time restrictions or because they did not fit into the overall concept anymore. On the other hand, new goals emerged during the many visits and calls with our commercial partner. One of these additional goals is the so-called re-evaluation functionality.

The issue of this feature is, to enable process management systems to appropriately react on changes to underlying data objects of processes. Business processes often have data objects as input and basis of their activities. Sometimes, these data objects cannot be locked during the execution of the process. This may lead to problems, when one of these objects is changed from outside the process. Currently, the whole process has to be stopped and is often completely thrown away and started again from the very beginning. This is, of course, a very disappointing situation.

Although this topic came up during the course of our project, it soon turned out to be too extensive for a bachelor’s project to deal with. A cooperation between the ‘Business Process Technology’ chair and SAP Research in Brisbane was set up to deal with the issue. This cooperation resulted, amongst other things, in the scientific paper “Re-evaluation of Business Processes”[3], that I worked on, too. In this paper, the scenario we had described and refined is used to introduce the issue and describe the basic mode of operation of re-evaluation. It also introduces the formal model for a solution of the issue, which we could implement inside the Petri net engine, that we had already worked on. The prototypical implementation and its description was my part in that paper, along with providing some advice for the formal model.

To explain the details of the prototypical implementation, I will first introduce the leading scenario for this issue. Afterwards I will sum up the important parts of the conceptual basis and the formal model described in the paper mentioned above[3]. Building on this information I will firstly describe the implications of re-evaluation on
the modeling environment and then explain the prototypical implementation inside the Petri net engine.

3.1 Leading Scenario - SRM Approval

The leading scenario for re-evaluation in our project was the following: Many large organisations currently use supplier relationship management systems to make their supply orderings more efficient and less expensive. These so-called SRM systems provide a portal for all employees and the company as a whole, where supplies, ranging from pencils, paper and whiteboard markers to office furniture and IT products up to raw material for the production process, can be ordered.

Apart from the economical advantage gained by these systems, they also provide some features for organizational tasks inside a large business. For example, purchase orders of employees can be put through an approval process, where the "purchase order may need to be approved by several roles, depending on the value and type of the purchase transaction, and on the ordering agent’s position in the organization’s hierarchy.”[3]

An example model for such an approval process is illustrated in Figure 3.1.

Figure 3.1: BPMN model of SRM approval scenario[3]
After an employee submits his purchase order, a confirmation e-mail is sent to him, availability of the ordered items is checked and corresponding products are reserved for this order. Then the approval process starts. For exemplary purposes we assume that the employee ordered new equipment for his office, namely a chair, a new desk and a laptop. His order needs to be approved by his responsible manager and because of the laptop, it also needs to go through IT-approval. An additional approval by the cost center is needed, because the total sum of his order exceeds 5,000€.

Now the employee wants to change the original order by replacing the laptop with a desktop computer. He might do so right after he submitted the order, but it is also possible that he does so only right before the order is actually submitted to the contractors. In these cases we need the possibility to determine, which tasks have to be re-executed because of the modified data.

3.2 Conceptual Model

In this section I will sum up the conceptual basis and the formal model of a solution to the re-evaluation issue given in [3].

Value-passing Nets

The algorithms and definitions are based on Value-passing Nets, which are a mathematical description for the kind of nets supported by our engine. They are very similar to Colored Petri nets with some additional features. One of the features are guard expressions, to restrict the enablement of transitions. Another one are filter-functions for read arcs, which can be used to express, that a transition only uses certain parts of an incoming token-value.

Transitions are enabled, when there is at least one valid token combination. This means that on each incoming place there is one token and the guard expression evaluates to true for this set of tokens. When a transition fires, it consumes the incoming tokens and produces one token per outgoing place. Figure 3.2 illustrates Value-passing nets by showing how the leading scenario model would be converted into this kind of Petri net. Note that value filters are not visualized there.

Re-evaluation algorithm

The high-level algorithm for re-evaluating such a net is fairly simple. Re-evaluation on the Petri net level is triggered by external manipulation of a single token. This forces the net to roll back until a state is reached, where the unmodified version of this particular token has no impact anymore. We can assure to have reached that state, when none of the transitions that have fired had this token as their input. After we rolled back we substitute the old token with the modified version. Now we can start rolling forward the net, by automatically re-doing transitions, whose filtered incoming token values haven’t changed critically compared to the previous values. If they really did change in a critical way, the transition has to be executed manually again.
Transition modes, execution log & execution context

To be able to roll back the net and provide the additional firing functionality mentioned above, we need to log and store the context in which transitions fire. The information we need are the following:

- transition name or identifier
- all input tokens, completely with their values
- all output tokens, also with complete values

We call such a triple a transition mode. Transition modes can be either active, which means that the mode represents the firing of a transition that has not been undone, or inactive. Inactive modes provide history data on previous firings of transitions, to be able to automatically re-execute transitions after a rollback.

The execution log is a collection of active transition modes, that are aligned by a half-ordering relation. This relation defines predecessors of transition modes, where one mode can have several predecessors and also be the predecessor of several modes. It needs to be a half-ordering, because of the parallel nature of business processes.
and Petri nets in general. Figure 3.3 shows an exemplary execution log based on the net in figure 3.2, like it would be after ‘Setup Support Plan’ was finished and the manager approval is about to be executed.

![Diagram of execution log](image)

Figure 3.3: Half-ordering of execution log

The execution context is an unordered collection, which contains only inactive transition modes, as opposed to the execution log.

**Firing with execution context**

If there are transition modes in the execution context for an enabled transition, then additional firing semantics are provided. We distinguish two cases: either there is exactly one mode, whose (filtered) input values are equal to the current ones or not. The second case is true, when there is either no mode with the same values or there are multiple ones with different output values. If there are multiple modes, that all have the same output values, then that’s part of the first case.

In the first case, we assume a functional dependency between input and output values\(^1\) and use the previous output values to automatically fire the current transition. In the latter case, we cannot assume such a functional dependency, because we either have no affiliated output values or there are different outputs for the same input values. Still, we can provide the previous input and output data to the user, when he re-executes the transition. This should enable him to quickly find out the differences in the input data and modify the output accordingly.

**Rollback**

Rolling back the net means reverting all transition modes that either had the token, which is being modified, as input or that are on the way from one of the ends of the execution log to one of the previous types of transition modes. For example, when looking at Figure 3.3 and assuming that only the IT approval had the modified token as input, then the IT approval and also the setting up of the support plan have to be reverted. Reverting here means taking the corresponding transition mode, deleting the created output tokens, recreating all input tokens and then moving the mode from execution log to execution context.

---

\(^1\)This is generally not the case in Value-passing nets, because of possible outside world effects like a manager changing his mind about an approval
Re-execution types & re-evaluation conditions

Additionally to value filters on read arcs, there are two more concepts for limiting the number of tasks to be re-executed and enabling the user or modeller to state more clearly how the system should behave, when re-evaluation becomes necessary. These concepts are introduced in [3] for a BPMN context, but for the purpose of implementation, I applied them to the Petri net world.

The first concept is the introduction of so-called re-evaluation conditions. These conditions specify, whether a transition needs to be executed again when the incoming values changed. It is for instance not necessary to manually execute the cost center approval again, if it was approved before and the total value of the shopping cart not increased.

The second concept distinguishes between different types of transitions concerning re-execution behaviour:

1. execute once: these transitions will never be re-executed, no matter how the data was changed. An example where this is useful would be 'Email process information’, because we already sent an email to the employee.

2. always re-execute: these transitions will always be executed again after they were rolled back. This should be used for 'Check availability’, because the requested items might not be available anymore even if the items themselves did not change.

3. execute conditionally: whether these transitions need to be re-executed or not depends on the modified data. If the filtered input values of such a transition did not change or the re-evaluation condition evaluates to false, then it does not need to be re-executed. In all other cases it is necessary to do so.

Restrictions

During the implementation of the algorithm described above, one drawback of this approach became clear. The problem is, that the algorithm’s current version does not support that a token to be modified externally, was modified internally before. Figure 3.4 illustrates the issue. Here I took the net from figure 3.2 and added one transition. This transition sets the status of the shopping cart to a certain value, say ‘in approval’, after the availability of all items was checked. This change in status causes the previous shopping cart token to be consumed and a new one to be created. If afterwards that new token is modified externally, the algorithm will not detect, that 'Check availability’ needs to be rolled back, because that transition had the old version of the shopping cart as input and not the current one. This causes the net not to be rolled back far enough, which may lead to inconsistent processes. In the illustrated case, assuming that a new item is added to the cart, the new item will not have been reserved and may not even be available, but will still be approved of and, if successful, ordered from the supplier.
So, in the current version of the algorithm, the modeller needs to beware that he should only use read arcs on critical places (or critical data objects, when talking about BPMN). The modeller does not only have to be careful because of re-evaluation, he can also use it to his and the process’ advantage.

### 3.3 Modeling Implications

The implementation of the re-evaluation feature is not limited to the execution engine. Instead, several concepts that we have seen in section 3.2 can be used by a modeller to affect how a certain process should react on changes of an underlying data object. In this Section I want to illustrate the implications of re-evaluation on a modeling environment. As opposed to the previous and next sections, I will put the focus on BPMN instead of Petri nets. I do so, because as I mentioned in Chapter 2 the basis for modeling processes in our environments is BPMN. In Galaxy, from the SAP AG, this is currently the only supported notation and in Oryx we restricted ourselves to BPMN, although it supports several other process modeling notations. So a potential modeller will need to know, what re-evaluation does on a BPMN level and also how he can affect its behaviour.

The need-to-know part for a modeller is summarized neatly in [3]:

Figure 3.4: Writing critical objects may lead to problems
When a change to an underlying data object happens, re-evaluation is triggered and then “carried out in three phases. (1) A decision whether re-evaluation is possible is made, and if possible, the roll-back phase begins. Otherwise, a modification to the data object is rejected (which may trigger human intervention should manual re-evaluation be possible). (2) In the roll-back-phase, the process instance is rolled back to a state where none of the executed process nodes (activities, gateways) is dependent on the data object. Then (3) the roll-forward-phase begins: the process nodes are analyzed and re-executed or skipped according to the rules.”

Re-evaluation Triggers

The triggering of re-evaluation is the first part that a modeller can and needs to specify. He can do so by using so-called re-evaluation events. [3] states that these events trigger re-evaluation based on changes to data object attributes. In our extension to the Process Composer in Galaxy we realized this behaviour not through explicit events, but by being able to state for each task, which incoming data objects or attributes are necessary for the execution of this task. A list of all necessary attributes can then be created when the model is being compiled. On the one hand this is probably a more intuitive way to define triggers for re-evaluation and it also guarantees that it is never triggered unnecessarily. This could happen with events, if a modeller defines an attribute within a re-evaluation event that is actually not needed, maybe because he does not know better or because he forgot to update the event after a change to the model.

On the other hand, this way of triggering re-evaluation could prove to be more time-consuming for a modeller, because he has to define the necessary attributes for each task in the model. This will surely be annoying if it has to be done for an already existing model or if it is always the same attribute. But for these cases one could provide the modeller with some convenience functions.

Re-execution types & re-evaluation conditions

The concept of different re-execution types for transitions explained in the previous section, comes from [3], where these types are introduced for BPMN tasks. Specifying the re-execution type for a task can be done easily by introducing a new options group for tasks, similar to ad-hoc or loop options. There the modeller can choose the specific type from a drop-down box or similar widgets. The default option should probably be ‘execute conditionally’.

Also within this options group, the modeller can customize a tasks re-evaluation behaviour by providing an expression that decides whether a task should be re-executed or not, if re-evaluation affects this task. This expression is optional, but can greatly improve the effectivity of the re-evaluation algorithm, since tasks may even be automatically re-executed although a necessary attribute has changed.
3.4 Prototypical Implementation

This section covers the details of how the re-evaluation functionality was implemented in the Petri net execution engine, which I already introduced in Section 2.4.3. Only one thing needs to be added to that introduction, which is the way the engine keeps its data. As can be seen in Figure 2.15 it uses ActiveRecord\(^2\) for accessing the underlying database. This is a plugin for Ruby, a so-called gem, which integrates access to database tables and entries natively into the Ruby language. It provides a framework, which makes it relatively easy to express and access relations between classes and have these relations represented in the database.

To give some examples, it is for instance possible to define a column in the database schema and, after migrating the database to that new schema, access that column as if it is an instance variable of the corresponding class. Similarly, when you want to express the relation that one execution log contains several transition modes, you just have to add the following line to the definition of ExecutionLog:

```ruby
has_many :transition_modes
```

This macro and the different others not only define relations, but also generate different accessor methods for manipulating instances belonging to such a relation. The number and type of methods depends on the type of relation. The above relation for example includes instance methods for getting, adding, removing and iterating over members of this relation.

Of course, in TransitionMode we also need to define the relation. Assuming that one TransitionMode can only be contained in one log we would write:

```ruby
belongs_to :execution_log
```

After adding the foreign key 'execution_log_id' to the schema definition of the transition mode’s table, we could start using the relation. We could for instance say:

```ruby
log = ExecutionLog.new
log.transition_modes << TransitionMode.new
log.transition_modes.each{|mode| puts mode.execution_log}
log.transition_modes.destroy
```

Having explained this, I will now go on and explain how the concepts shown in Section 3.2 were implemented.

3.4.1 Transition Modes

The first thing to get working, were the transition modes, because they are at the hub and center of the whole algorithm. But prior to this, a more immediate problem had to be solved.

\(^2\)http://ar.rubyonrails.com or http://rubyforge.org/projects/activerecord


**Consuming tokens**

One major problem could have been that transition modes have to store the complete content of the tokens they consumed and created, to be able to restore them later on. Tokens and their values would have to be moved to different tables whenever they were consumed or recreated, which would have led to serious data traffic inside the database, whenever a transition fired and especially when re-evaluation would be needed. The solution here is conceptually rather easy and was brought up by our supervisor Gero Decker. The idea is to not delete tokens when they are consumed but rather just set them inactive, so that the engine ignores them concerning firing and enabling. That way we have virtually no traffic in the database and can still reference the tokens in the transition modes.

The implementation of this feature affected three places in the code. The first was the `Token` class, to which a new instance variable named `is_active` and corresponding getter and setter were added.

To realize that token are not destroyed anymore when they are being consumed, the `consume` method inside the transition class had to be modified accordingly.

Now, that consumed tokens remained inside the database we had to make sure, that only active tokens are regarded by the engine concerning enablement and firing. The easiest way to do this, was to manipulate the view on tokens lying on a place, making sure that only active tokens are displayed to other classes. Since places are the only way through which tokens are being accessed by other classes it was sufficient to modify this one class. However, this proved to be more difficult than expected.

Places have a one-to-many relation with tokens, which is realized by ActiveRecord macros. In the place class this is: `has_many :tokens`. The macro generates amongst other a method `tokens()`, which is the one access point to tokens used by other classes. In order to return only the active tokens, it seemed that we had to modify that auto-generated method. This is not trivial, because that method never appears in the code. With some advanced Ruby features, though, we managed to rename that method to `old_tokens()` and define a new `tokens()`, which called the old one and selected only the active tokens. This solution worked fine in the way that it only returned active tokens lying on a certain place, but from that moment on all other auto-generated methods stopped working properly. A probable explanation could be that `old_tokens()` returned the tokens in a certain way, maybe with some additional information, that the other methods relied on and which got lost when selecting only certain tokens out of the collection. Whatever the problem was, we could not use that solution. After some research and several trials and errors, an actually rather simple solution was found.

One of the many options for the macros, is the so-called finder-sql, which is an SQL-statement for specifying, which entities should be regarded by this special relation. The following lines solved the problem of returning only active tokens from a certain place:
3.4. Prototypical Implementation

```ruby
has_many :tokens, :finder_sql =>
  'SELECT * FROM tokens ' +
  'WHERE tokens.place_id = #{id} AND tokens.is_active = true '
```

Relations to other Classes

Now we could start implementing transition modes. In section 3.2 we said that a transition mode needs the transition it belongs to, together with the input and output tokens. The implementation of the according relations is rather simple, with one exception: One transition mode can have multiple input and output tokens. But while one token can only be the output of one mode (one-to-many relationship), one token can be the input of several modes (many-to-many), because we support the concept of read arcs, that don’t consume tokens when the transition fires.

When implementing transition modes inside the engine, we need some more information, though. One transition can be contained in an execution log as well as in an execution context, but only in one at a time. To represent this, we introduce a polymorphic many-to-one relation to an 'execution_container', which can be either an instance of ExecutionLog or ExecutionContext.

Also, we gave the responsibility of knowing the preceding modes of a transition mode to the modes themself, instead of requiring it from the log. We did this, because each mode already holds the information needed to find out its predecessors in the form of their input tokens. It is a fact that each token is created during one specific firing of a transition. Since this firing of a transition is represented by a certain transition mode, we can say that each token is created by one transition mode. In that mode, the token appears as the mode’s output token. By using that information, a mode can easily find out its predecessors, by getting the creator modes of its input tokens. To make this knowledge explicit and ease the traversal of modes inside an execution log, we define two relations between transition modes, the preceding modes and the succeeding modes relation.

Functionality

The 'TransitionMode' class also provides some useful functionality. The screenshot of a transition mode displaying all information belonging to that mode can be seen in figure 3.5. Usually, also the values of the tokens are displayed, but this feature has been disabled for the sake of clarity for the making this screenshot.

Firstly, it determines the predecessors only the first time that they are needed. Before that time the information exists only implicitly and afterwards it is stored explicitly in the precedence relations mentioned above.

Also, transition modes can be undone, which reverts the effects they had on the net. This implies deactivating all output tokens, activating all input tokens and moving the mode into the execution context. Of course, this can only be done,
when all output tokens are active beforehand and all input tokens inactive, with the exception of tokens that were only read by the transition.

Similarly, when all input tokens are active and all output tokens inactive, a mode can be re-done, which also causes it to be moved into the execution log again. This functionality cannot be used for re-evaluation, because when re-executing the input tokens will not be the same ones as before, but it makes debugging and testing a lot easier.

### 3.4.2 Execution-Log and -Context

An *execution log* is a collection of transition modes, which are aligned by a half-ordering relation, the precedence relation. Since the implementation of that relation
was moved to the modes themselves, it is not needed by the log anymore. However, it provides some other methods to support the re-evaluation of processes.

The first thing to mention is, that the engine we worked on was already modified by us to support the notion of a case, which is the representation of a process instance. Usually such a concept is not found in a Petri net world, but since it was already present, we could use it to our advantage. We defined execution logs to belong to one specific case, thereby limiting the effect of needed re-evaluation to one case, instead of affecting the whole net.

Having this separation of logs on case also simplified the implementation of other functions, like determining whether modes have a certain mode as their ancestor in the log. A mode is a predecessor of another one if they have a direct connection within the log and he is an ancestor of that mode if he is the predecessor of a predecessor and so forth.

Also the functionality of rolling back a case is implemented within the execution log. When a token is modified externally, we determine the token’s case and call the method `roll_back(modified_token)`. We then find out all modes that had this token as an input token, which is an easy task, because we already have that information in the database by means of the input_tokens relation of transition modes, which is known as the depending_modes relation inside the token class. Also determine the modes that present the ends of the execution log and store them as an array in `last_modes`. Having the list of the critical modes, we now roll back the net, by taking the first mode from `last_modes` and determining whether he has any of the critical nodes as his ancestor or is one himself. If this turns out to be true, we undo the mode, add his predecessors to the `last_modes` array and removing himself from it. Now we repeat this procedure until all critical nodes have been undone.

The counterpart of the execution log, is the execution context, which is a set of all undone transition modes of one case. As such it represents a central knowledge repository of one case, from which results of previous activities can be extracted. To be able to easier do so, it provides some methods for finding transition modes belonging to certain transitions or having input tokens that are equal to the current ones.

3.4.3 Enhanced Firing Semantics

When a case was rolled back and the execution context is full of transition modes, we use these undone modes to help the process to be re-executed more quickly. We do so by querying the context for transitions that are currently enabled. If we find one, we test whether the old input token had equal values as the current ones. If that is the case, we take the old output tokens and use them as a template for creating new output tokens for this transition. If the values are not equal, we provide the user having to re-execute the task with the old input and output values, highlighting the values that have changed. This feature is illustrated using a screenshot from a conceptual prototype, because unfortunately that feature was not
3. Re-evaluation of BPMN Processes

completely finished at the time this thesis was written. The screenshot can be seen in figure 3.6.

![Screenshot of task to be re-executed](image)

Figure 3.6: Screenshot of task to be re-executed

Re-using values that have not changed, he might be able to execute the task quicker.

To realize the automatic re-execution of tasks, whose inputs have not changed, we extended ‘notify_following_transitions()’. This method is called after each firing of a transition, to trigger automatic transitions located behind this one. Before the ‘trigger_fire()’ method for automatic transitions is called, we now first call ‘trigger_reexecution()’. There we check, if there is a transition mode in the context that represents a firing of that transition. If that is the case and if the current input tokens have equal values to the previous ones, then we trigger the automatic re-execution of that transition. This involves consuming, and thus deactivating, the current input tokens, getting the values of the previous output tokens and putting these values inside new tokens, which we put onto the corresponding places.

If current and previous values are not the same, then we store previous in- and output values for providing them to the user later on. To do so, we needed to get these values to the user interface, which required a change in the XForms template. It was extended with an input instance containing the previous token values, if there are some.
Using the implementation, the rolling forward of a net is not a secluded phase or algorithm that is executed, like the rollback phase. Instead it just provides the additional functionality for enabled transitions.

When the current transition is an automatic one, we actually would not need to provide any of the extended execution semantics. If the input values are the same, then the automatism will create the same output anyway and the provision of previous data is unessential, since it cannot take any advantage of it. It still may be faster to try to automatically redo such a transition, if the search for a transition mode with equal input data is reasonably fast, compared to the average execution time of automatic transitions. Since in our implementation, there is one execution log per case, that log will usually be rather small and will in most situation contain at most one transition mode per transition. That’s why we provide the automatic re-execution for automatic transition, too.

3.5 Outlook

After I explained what we implemented and how we did this, one can see that this was only a first step to explore this issue. Unfortunately we could not realize all concepts described in section 3.2 in our prototypical implementation, because of time and resource constraints. For example, the features of value filters for read arcs, re-evaluation conditions for transitions and re-evaluation events or necessary attributes respectively are missing. As soon as the latter feature is implemented we can handle not only external changes on data objects, but also internal ones. The possible advantages can be illustrated with the help of the ‘Hire-an-Employee’ scenario introduced in section 2.1. Lets assume, that the employee information filled in has to be approved by the responsible manager at some time within the ad-hoc subprocess. If the information is rejected and must be changed after a homepage and the workplace have already been set up, then we could use the re-evaluation functionality for checking, whether these two tasks have to be done again.

There are some things missing, though, even on the conceptual side. I already illustrated one drawback of the current algorithm in section 3.2, but there are others. One problem of the current solution is, that it bases on Petri nets, which have a closed-world semantic. This is an assumption stating that all information consumed, modified or created by a transition is represented inside the net. Unfortunately this assumption does not always hold. The ‘Submit order’ transition, for instance, has a side effect on the supplier that is not represented in our Petri net and cannot just be reverted. Either we must specify such transitions as ‘not revertable’ or we must implement concepts for executing compensating activities.

Also, the effects of re-evaluation on advanced modeling constructs like ‘Multiple Instances’ or the effects on process choreographies need to be researched.
4. Conclusion

This chapter sums up the work done for the two areas described in this thesis and also on the bachelor’s project as a whole.

Bringing BPMN to Execution

In this part of the thesis, I presented concepts for translating BPMN constructs into Petri nets, that behave like their corresponding BPMN element when executed. I also showed, how we implemented the translation inside the Oryx modeling tool and what changes were needed on the Petri net execution engine to support the resulting nets.

The number of BPMN constructs we currently support is not even close to the total number of constructs defined in the specification. However, this was not the goal of our project, since we concentrated on finding and realizing constructs for supporting ad-hoc processes. The translation algorithm we built on, is generic enough so that the missing constructs can be added later on.

The notion of the context place we introduced, provides a clean separation between types of data inside a task and a single-point of access for getting metadata at runtime. It’s realization also provided valuable experiences and implementation constructs, which could be re-used for the conversion of data objects and the management of data places inside the engine. Also, the concept of the context places could be extended by enabling other tasks to access this information at runtime. Currently some context-XSLTs assume, that there is only one context place used by the transitions within a task subnet. Removing this restriction could enable such advanced security concepts like the four-eyes principle, making sure that certain different tasks are not executed by the same person.

Re-evaluation

The re-evaluation functionality implemented during this project is a powerful feature, that has never been implemented before. It enables processes to adapt to external disturbing factors much better than in current systems.
With some more work on the implementation this could also be extended to internal factors. That way, faulty models or problems with parallelism inside one model could be handled without throwing away whole process instances.

The current implementation of this feature even provides advantages when no re-evaluation is needed. Execution logs, for instance, provide a great overview on the history of a current case. Together with the transition modes, one can check whether the right tokens were consumed or created, and also which input lead to which output values. This turned out to be a great help for developing transition- and arc-transformations. Additionally, the feature of undoing and redoing transition modes is a great tool for testing and stepping through the deployed Petri nets.

Of course, there are also some disadvantages. Currently, there are many redundant execution information inside the prototype. For instance, the firing of a transition in a task subnet is logged in three different ways: through a transition mode, as an 'action' tag inside the task’s context place and in the CSV logger. Also, the memory usage in the database can possibly be very high, if we have many long-running or complex processes, since all tokens ever created during a case are stored as long as that case is not finished. Also, the feedback for possible users is rather bare-bone in our prototype. Questions of user acceptance have to be regarded and also the possible confusion of users, when some tasks have to be done again and others mysteriously not.

To sum up, I think the advantages of re-evaluation and its current implementation outweigh the disadvantages by far. If this should not be the case in a special situation, the whole functionality can be easily toggled on or off through a single configuration option, eliminating all disadvantages I just mentioned. In the end, re-evaluation makes working with business process management tools much easier, even in its current prototypical form.

**Bachelorsproject**

During this bachelor's project we spent some time with activities that are not be represented in this bachelor's thesis and sometimes don’t even appear in any of the theses. An activity for the first case was the process of defining terms concerning ad-hocness. The latter case includes extracting and refining use cases, that in the end were dropped again. But still, this work also was important, because now we have a clearer understanding of ad-hoc processes and we also know, which use cases not to take as a basis for further research. However, through the use cases we documented, the prototypes we developed, the extensions we made on different tools for the process lifecycle and through integrating these tools to form a single process management system, we provided the base for future projects to build on our results. Additionally, we delivered the proof that the concepts we chose to support, can not only be modeled, but also be brought to execution and used in real systems.
Bibliography


