

Bachelor's Thesis

# Modeling and Executing Ad-Hoc Subprocesses in Different Environments

Stefan Krumnow

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 run time. This bachelor's thesis 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.

The thesis concentrates on Ad-Hoc Subprocesses that can be used to model flexible run-time behavior. Therefore, definitions and examples are given that classify the performed work. Since two different modeling environments have been used, these tools are compared. Moreover, it is shown, how they have been extended with the possibility to model completion conditions that define when an Ad-Hoc Subprocesses completes its execution. Finally, the transformation of an Ad-Hoc Subprocess into a Petri net that can be executed is explained in detail.

## **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 gefundenen Lösungen zu veranschaulichen.

Die Arbeit konzentriert sich auf Ad-Hoc Subprozesse, die modelliert werden können um flexibles Laufzeit-Verhalten auszudrücken. Zur Einordnung der durchgeführten Arbeit werden Definitionen und Beispiele gegeben. Da zwei verschiedene Modellierungsumgebungen genutzt wurden, werden diese in der Arbeit verglichen. Außerdem wird gezeigt, wie die Umgebungen um die Möglichkeit der Modellierung von Abschluss-Bedingungen, welche definieren, wann ein Ad-Hoc Subprozesse seine Ausführung beendet, erweitert wurden. Abschließend wird die Transformation von Ad-Hoc Subprozessen in ausführbare Petri Netze detailliert erklärt.

# Contents

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Introduction</b>                                    | <b>1</b>  |
| 1.1      | Bachelor's Project . . . . .                           | 1         |
| 1.2      | Work on Design-Time based Adhocness . . . . .          | 2         |
| <b>2</b> | <b>Definitions</b>                                     | <b>5</b>  |
| 2.1      | Adhocness . . . . .                                    | 5         |
| 2.2      | Example Scenario: Rate Job Applicant . . . . .         | 7         |
| 2.3      | Design-Time-based Adhocness . . . . .                  | 8         |
| 2.4      | Completion Conditions . . . . .                        | 11        |
| <b>3</b> | <b>Comparing Galaxy and Oryx</b>                       | <b>13</b> |
| 3.1      | Comparing Architectures . . . . .                      | 13        |
| 3.2      | Comparing the Modeling Tools . . . . .                 | 15        |
| 3.2.1    | Features . . . . .                                     | 15        |
| 3.2.2    | Architectures . . . . .                                | 16        |
| 3.3      | Implementation of Completion Conditions . . . . .      | 19        |
| 3.3.1    | Process Composer . . . . .                             | 20        |
| 3.3.2    | Oryx . . . . .   | 22        |
| 3.4      | Survey of Changes in Oryx . . . . .                    | 23        |
| <b>4</b> | <b>Transformation</b>                                  | <b>25</b> |
| 4.1      | Transforming BPMN Activities into Petri nets . . . . . | 25        |
| 4.1.1    | General Activity Lifecycle . . . . .                   | 26        |

---

|          |  |           |
|----------|--|-----------|
| 4.1.2    | Transformer Implementation . . . . .                         | 27        |
| 4.2      | Transforming Ad-Hoc Subprocesses into Petri nets . . . . .   | 27        |
| 4.2.1    | Structural Transformation . . . . .                          | 28        |
| 4.2.2    | Transformation of Completion Conditions . . . . .            | 31        |
| 4.3      | Transforming Dependency Constructs into Petri nets . . . . . | 32        |
| <b>5</b> | <b>Conclusion</b>  | <b>35</b> |
| 5.1      | Ad-Hoc Subprocesses . . . . .                                | 35        |
| 5.2      | Bachelor's Project . . . . .                                 | 36        |
|          | <b>Bibliography</b>  | <b>37</b> |

# 1. Introduction

In recent years, workflow management systems have gained much influence. By using process models as input for a generic workflow engine, development and execution of business processes is simplified. Especially well structured and highly repetitive processes are supported well by existing solutions [Wes07]. Thereby, automated activities can be combined with manually executed ones.

But there are also processes that cannot be pressed into such rigid structures. Those creative processes are executed by human workers that contribute a lot of individual knowledge and experience. Process instances differ not only in data but also in structure and number of participants.

Due to their rigid process structures, traditional workflow management systems cannot cope with those processes properly. Neither the modeling tools nor the execution environments provide enough flexibility to support them.

## 1.1 Bachelor's Project

The bachelor's project, this thesis is based on, examined degrees of ad-hoc changes that shall be supported by workflow management systems in order to execute more flexible processes. Together with SAP AG and the team at the chair of Prof. Weske, use cases were analyzed, concepts created and prototypes developed.

SAP AG is currently developing a new workflow management suite based on the *Business Process Modeling Notation* (BPMN [bpm08]). The product "NetWeaver Business Process Management" (code name: *Galaxy*) shall cover traditional workflows as well as ones with more flexible behavior. Therefore, the identification of relevant concepts is an important issue in order to integrate related functionality in future implementation cycles.

Besides the SAP product, the *Oryx* platform [DGKW08] was used as second workflow system. This platform is a scientific open-source solution that can be extended

easily. It bases on a modeling tool developed in a previous bachelor's project [Tsc07]. Again, BPMN is one of the supported modeling notations.

In early phases of the project, related work in the area of flexible workflows was analyzed and use cases were examined. Therefore, we 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 participated in meetings and calls.

During the project, three major concepts were identified: Besides design-time based adhocness that is supported by new modeling constructs, delegation and re-evaluation have been studied. Delegation can be used to interact with one or several other workers at run time. Thereby, the interaction becomes part of the executing workflow system. Re-evaluation functionality helps to react on changes in required data objects automatically.

Prototypes supporting the identified concepts were implemented, both within Galaxy and Oryx. Due to its complex Process Server, Galaxy was only extended with design-time features, though. The run-time semantics of the implemented constructs has been examined using prototypes in the Oryx platform which was extended with modeling and execution functionality.

Besides two theses in the context of design-time-based adhocness, four more theses have been written: An overview on identified use cases and the resulting usability considerations is given in [Kog08]. [Nag08] shows the concepts and implementations in the area of delegations while [Mas08] focuses on re-evaluation and model transformations. In [Ger08] changes on Oryx are specified and concepts for process analysis are explained.

## 1.2 Work on Design-Time based Adhocness

Beside delegation and re-evaluation, design-time-based adhocness has been a major concept. Here, new constructs are introduced that enable the modeler to create more flexible process models in an intuitive manner. For this, the concept of a BPMN *Ad-Hoc Subprocess* has been taken up as well as scientific approaches such as *Case Handling* [vdAWG05] and *Pockets of Flexibility* [SSO01].

The thesis [Kle08] introduces important use cases and derives necessary constructs from them. Moreover, an overview on all changes performed on the Galaxy modeling tool *Process Composer* is given. While [Kle08] focuses on design-time constructs and the implementation within Galaxy, this thesis also regards changes on Oryx and execution aspects.

The remainder of this thesis is structured as follows: Chapter 2 explains necessary terms and issues. Thereby, the term *Adhocness* is defined before an example and useful model constructs are introduced and explained. The chapter mainly represents early stages of the bachelor's project.

In chapter 3, Oryx and the Galaxy modeling tool Process Composer are compared. Therefore, the implementation of ad-hoc completion conditions in both environments is shown exemplary. Moreover, a survey of changes that enable the modeling of Ad-Hoc Subprocesses in Oryx is given.

Chapter 4 explains how the modeled Ad-Hoc Subprocesses can be executed in Oryx. For this purpose, they have to be transformed into Petri nets. After the general BPMN to Petri net mapping is introduced, the transformation of Ad-Hoc Subprocesses is explained in detail.

Finally, chapter 5 gives a conclusion by regarding the work in context of this thesis as well as of the whole project.

## 2. Definitions

This chapter explains terms and issues that are required to understand the thesis' remainder. Thereby, it mostly reflects early parts of the project that dealt with theoretical classification, the elaboration of use cases and the creation of solving concepts.

Subsection 2.1 presents a general explanation and classification of the term *Adhocness* that bases on related work. Then, subsection 2.2 introduces an exemplary scenario that is used throughout this thesis. In 2.3 essential characteristics of design-time-based adhocness are explained referring to [Kle08]. Finally, subsection 2.4 examines a special design-time constraint, the completion condition, more closely.

### 2.1 Adhocness

The term *ad-hoc* is rather unexplored in the area of business process management. There is some research that works on flexible and dynamic issues but there is no general understanding of *Adhocness*. So, in order to determine a definition or explanation of the term, related work in the field of flexibility has to be regarded as well as general language usage.

In [SSO01], Shazia W. Sadiq et al. try to classify different dimensions of change that can occur in workflow management. They identify 3 basic properties a system can have: Dynamism, Flexibility and Adaptability. Dynamism describes the ability of a system to change and migrate the model of a single process instance or of all instances.

Adaptability stands for the feature of reacting to exceptions that occur during the execution of a process instance. Those may be foreseen and, therefore, specified in the process model. But there are also exceptions that cannot be anticipated and have to be handled at run time only.

Flexibility describes that loosely or partly specified processes may be modeled. Instances of those models must complete the full process model at run time. Therefore, every instance may take an unique execution path in a potentially unique model. The crucial question at this point is, what a *loosely or partly specified model* is. The authors argue that a even a huge number of choice constructs (such as XOR gateways in BPMN) can create a loosely specified model.

There are many ideas on how to create flexible behavior in workflows. Besides their classification, Sadiq et al. introduce a construct called *Pocket of Flexibility* that contains a number of subtasks without a complete order. The idea is very similar to *Ad-Hoc Subprocess* specified in BPMN [bpm08].

In [AtHEvdA06], the authors present *Worklets* as exchangeable and reusable components that can be integrated at specified points in a running instance. This approach is both, flexible and adaptive, since the creation and usage of worklets can be used to react to exceptions.

In general language usage, the term ad-hoc means *"for the specific purpose, case, or situation at hand and for no other"*. Transferring this to the world of workflows, ad-hoc behavior occurs during the execution of single process instances: An additional activity may be desired, another worker needed or a different execution order required. In terminology of [SSO01], ad-hoc behavior could support an adaptive or flexible system. Since dynamism aims for the creation and modification of process models, it will no longer be regarded in this thesis.

As described before, adaptability means the ability of a system to react to exceptions. The handling of unanticipated exceptions at run time needs to be done ad-hoc. Special workers could be invited to join the execution of a task using delegation mechanisms. Components such as Worklets could be loaded ad-hoc in order react individually. Those feature are purely run-time sided.

But there are also things that need to be done at design time, e.g. the definition of compensation and re-evaluation features. Due to the fact, that a run-time user most likely has not enough technical knowledge to specify a compensation behavior, this should be done at design time. However, during run time, those features can be used ad-hoc.

Another example in this area of conflict is the ad-hoc creation of new tasks at run time. The majority of users could not cope with this situation. Therefore, the modeler has to create a flexible structure in which the user can work ad-hoc: He could decide to use one out of several possible ways to deal with a situation by choosing a Worklet out of a repository. Another solution is to execute task instances within an Ad-Hoc Subprocess in an order that suits best.

Inferentially, there seem to be two types of adhocness that need to be regarded. Purely at run time realized adhocness like delegations and design-time-based adhocness defined by a modeler but triggered and used by the run-time worker. This thesis will focus on design-time-based adhocness.

## 2.2 Example Scenario: Rate Job Applicant

In this thesis, a simple example process is used to show how insufficient traditional workflow management systems can be if certain conditions are met. The process derives from the area of human resource management and describes how a potentially new employee is ranked and possibly hired.

The evaluation of a job applicant starts by reading his application and his CV. If those documents are interesting for the human resource department, there are a number of potential steps that can be taken in order to check his capability:

- A human resource manager could try to find additional material about the applicant on the internet, e.g. by using Google. The search results could be used in future activities.
- The applicant could be invited for an interview. Since he has to appear on a specific time, the invitation is a precondition for the interview itself. The agreed date and time should be filed using the workflow system.
- After he has been invited, the actual interview is possible. Afterwards, a protocol or maybe a video-tape needs to be archived.
- If the interview is videotaped it might be analyzed later. This is useful, especially if open questions remain or an important person could not attend the interview. After the analysis an evaluation is saved.
- A telephone call could be used as a pre-interview as well as a substitution of a real meeting, e.g. if the applicant lives abroad. Moreover, it could be done after the personal interview in order to resolve open issues.
- When enough information is gathered in order to decide whether the applicant should be hired or not, this decision needs to be made. If he/she should be hired, another process will start that is responsible for making a contract and setting up a workplace.

This process contains a lot of activities that do not need to be executed, but can be executed, if the worker thinks they are necessary. Also, different tasks could be done by different users, e.g. an intern could search the applicant on the internet while a secretary invites the applicant and a manager performs the actual interview.

In case of a student that wants to attend an internship abroad, a personal meeting would be too expensive. Hence, the human resource manager only tries to find information about him using Google after he read his application. Next, he decides that a telephone call is necessary to find out, whether the student should be hired. After the call, the manager thinks that the student is capable for his company and submits the decision to hire him. Since there was no interview, no invitation needed to be made and no video analysis could be executed.

Considering the case of a new manager, the executed process gets more complicated. Naturally, he is called for a first talk after his references have been read and checked using the internet. After the call, he is invited for an interview. Although the meeting is taped, the tape does not have to be viewed again, since the interview reveals that the applicant is not capable for the advertised job. Therefore, the decision is made that he will not be hired which completes the process.

Unfortunately, traditional workflow systems are not eligible for modeling such a process. In BPMN, there are several problems:

- In order to express that a task does not have to be executed, event- or data-based decision XOR gateways must be used.
- Since no other activity should be executed after the decision whether the applicant should be hired or not was made, *"Submit Decision"* must be placed last. Therefore all other tasks have to be completed or somehow skipped before this task can be executed.
- If one wants to express that two tasks can be executed independently and in any order, an AND gateway can be used.

Figure 2.1 shows how the process in BPMN looks like. It should be regarded that simple data-based XOR gateways are used in order to express the user's decision on whether to execute or to skip tasks. The gateway ahead of *"Analyze Video"* does also express, that a video was taped before. Moreover, no pools or lanes are used and only the video but no further data objects are modeled, for the sake of brevity.

The following two subsections present constructs that enable the modeler to design this process more intuitive and simple.

## 2.3 Design-Time-based Adhocness

Besides very rigid business processes that run often and are highly automated, there are processes that require a lot of human creativity during execution. Those processes cannot be foreseen fully at design time which results in process models that will restrict creativity at run time due to rigid constructs or are ugly due to many gateways. The example shown in figure 2.2 describes such a process.

There are already a number of concepts that enable the modeler to create more flexible diagrams although none of those concepts was realized in a big commercial workflow engine.

The BPMN specification [bpm08] describes an Ad-Hoc Subprocess that can be modeled by using an embedded subprocess and setting the *isAdhoc*-attribute to true. Ad-Hoc Subprocesses are defined as follows:

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

An Ad-Hoc Subprocess has two attributes that must be set: An *AdHocOrdering* attribute defines whether the sub activities must be executed sequentially or can be executed parallel. An *AdHocCompletionConditions* specifies, when the execution of the Ad-Hoc Subprocess completes and the successor activity becomes activated. This concept has not been realized in any BPMN execution engine, yet.

The already mentioned Pockets of Flexibility are similar constructs. They also can contain a set of not ordered subactivities that can carry interdependencies. It is not specified how these dependencies may look like or how the pockets can be executed exactly. Within SAP Research, the concept is now further examined under the name of *Flexible Blocks* [BGK<sup>+</sup>07].

In order to define dependencies in such environments, concepts that do not use iterative sequence flow should be regarded:

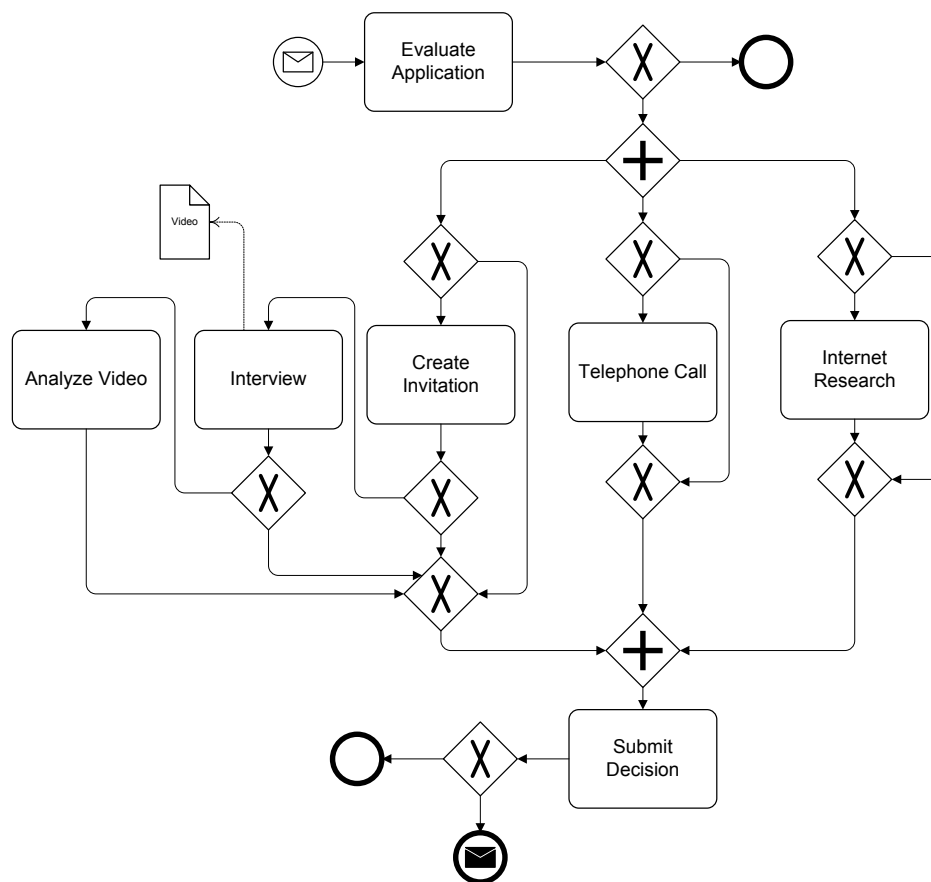


Figure 2.1: Example Process in Traditional BPMN 1.1 (Simplified)

One idea behind *Case Handling* [RRvdA03, vdAWG05] is to use data dependencies instead of or besides control flow. Activities may be executed if all their input data was written before. By modeling data dependencies, the functionality of an activity gains more focus instead of its placement. Moreover, the run-time user should get an overview of the context his/her activity is located in. Thereby, his understanding of motives and conditions is increased which make the execution of (especially creative) business processes more efficient.

The paper [PSvdA07] introduces a declarative modeling notation (*DECLARE*) that allows to model conditions such as "After one execution of *A*, *B* will be executed at least once", "If *A* is executed, *B* was executed before" or "A may be executed only once" using linear temporal logic. The system is integrated with YAWL and can, therefore, be used with iterative models, too.

After examining all those concepts, we started to develop an ad-hoc environment that can be used to model creative sub processes. Thereby, we are using the weak BPMN specification of an Ad-Hoc Subprocess. Since only BPMN Tasks are defined to be potential subelements of an Ad-Hoc Subprocess, the specification needed to be extended.

In figure 2.2, the example process from section 2.2 is modeled again using an Ad-Hoc Subprocess. Here, also sequence flow and data dependencies are used to express relations between tasks. The sequence flow between "Create Invitation" and "Interview" models, that the interview cannot be executed if no invitation has been send. The data dependency from "Video" to "Analyze Video" means that the task requires a video that is not null in order to start. Again, except from the video, no data objects are modeled in this diagram.

Generally, we decided that sequence flow and data dependencies are needed to extend the subprocess usefully. Temporal pre- and post-conditions, as used in *DECLARE*, did not seem to be intuitive enough to be used in a BPMN environment. Unlike the BPMN specification, we assume that all contained subtasks can only be executed once, as long as no loops or multiple instance markers are used. This enables the modeler to restrict the run-time behavior by using already known constructs. Moreover, the standard completion of an Ad-Hoc Subprocess, as shown in section 2.4, gets a clear semantic by this decision.

In [Kle08], a further use case for design-time-based adhocness is introduced and required constructs as well as unnecessary ones are examined in more detail.

By using the Ad-Hoc Subprocess the excessive structure of undetermined execution order and optional tasks in diagram 2.1 is reduced to a compact construct with simple semantics. Only the placement of "Submit Decision" as completing activity remains to be explained. This will be done in the next section.

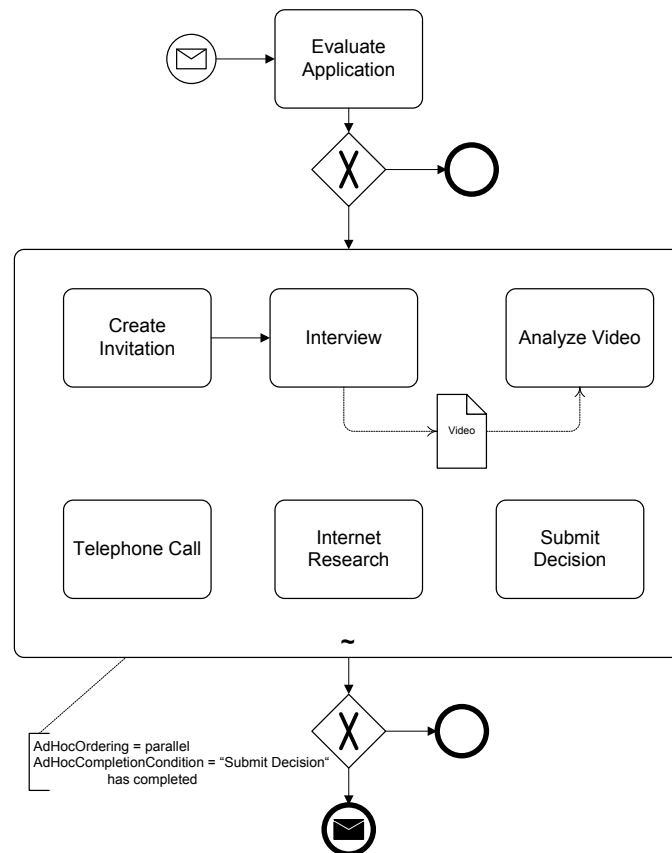


Figure 2.2: Example Process modeled using an Ad-Hoc Subprocess

## 2.4 Completion Conditions

The BPMN specification defines that an Ad-Hoc Subprocess must have a completion condition:

”If the Process is Ad Hoc (the AdHoc attribute is True), then the Ad-HocCompletionCondition attribute MUST be included. This attribute defines the conditions when the Process will end.”

There is no further definition on what kind of data should be checked within the condition, when the checks are performed or what should happen to running tasks when the completion condition evaluates to true. These questions need to be examined.

In principle, one can argue that all kinds of data available during the execution of an Ad-Hoc Subprocess should be usable in a completion condition. But since special meta data eventually has to be created for this purpose only, it’s worth thinking about it. Besides the data that is modeled at design time, also execution states of tasks should be checked. So, in the process model shown in figure 2.2 the execution state of *Submit Decision* can now be used within a completion condition. That

solves the problem of finishing the subprocess when a decision was made without having to skip all remaining task manually before.

We could not identify any use cases for checking other meta data such as time, date, executing owner or responsible roles within a completion condition. Moreover, the usage of such data might lead into problems, e.g. if one consider the case of a hardcoded time referencing the past.

Basically, the check for completion should be triggered every time the state of an task or the content of a data object changes. If the condition evaluates to false, the execution of the Ad-Hoc Subprocess should proceed normally. If it evaluates to true, the Ad-Hoc Subprocess should complete its execution.

In a parallel ordered ad-hoc subtask, it is possible that several subtasks are running, when the completion condition check returns true. Those tasks must be allowed to complete their execution since an partially performed task may leave data in a inconsistent state. Letting other tasks finish their execution leads to the problem that one of those tasks might change states that let the condition evaluate to false again. How to deal with this problem results in an important design decision that will be regarded later.

Problems can occur if the completion condition is inconsistent in a way, that there are cases in which it will never evaluate to true. E.g., if the condition checks the content of a data object it is not guaranteed that the required value will ever be written into it.

Therefore a standard safeguard completion condition should be implemented that completes the subprocess when all contained tasks are finished. That requires that *Loop Activities* distinguish between finishing one pass and finishing the whole activity. This standard condition does also provide an convenient default attribute value, since the BPMN specification demands that a completion condition has to be set.

## 3. Comparing Galaxy and Oryx

After looking into theoretical backgrounds, the used technology will be introduced, now. The project worked on two different platforms. SAP's new generation Business Process Management Suite (code name: *Galaxy*) was extended with design time features allowing to model more flexible processes. Execution semantics for these features was implemented within the Oryx platform that was developed at the chair of Prof. Weske at the HPI. Therefore, also Oryx had to be extended by new modeling constructs.

This chapter compares the two platforms focusing on the modeling environments. For this purpose, sections 3.1 and 3.2 compare the architectures of the whole platforms and the modeling tools. Section 3.3 shows how new modeling constructs can be implemented in both solutions by the example of completion conditions for BPMN Ad-Hoc Subprocess. Finally, section 3.4 gives a survey over all changes that were necessary to model Ad-Hoc Subprocesses with Oryx.

### 3.1 Comparing Architectures

Both, the Galaxy platform and the Oryx platform, are developed in order to support process modeling as well as process execution. Galaxy is supposed to be the new main workflow management solution within SAP AG. It will aggregate the functionality of several older solutions and should displace them. Therefore, it has to cover many different requirements. Most important are performance and security aspects, since Galaxy's process execution engine will be used in production systems of SAP's customers.

The Oryx platform, named after its modeling tool that was built first, is developed in a science project. It focuses on modeling and prototypic execution of business processes in the world wide web. Due to those different usage scenarios, Oryx' structure is more flexible and the platform is easier to extend with new functionalities.

Nevertheless, the high-level architectures of both platforms are equal to each other, as figure 3.1 shows. Both platforms have a modeling environment, called Process Composer and Oryx, that enable modelers to create and to manage process models. Those tools are regarded more detailed and compared in sections 3.2 and 3.3.

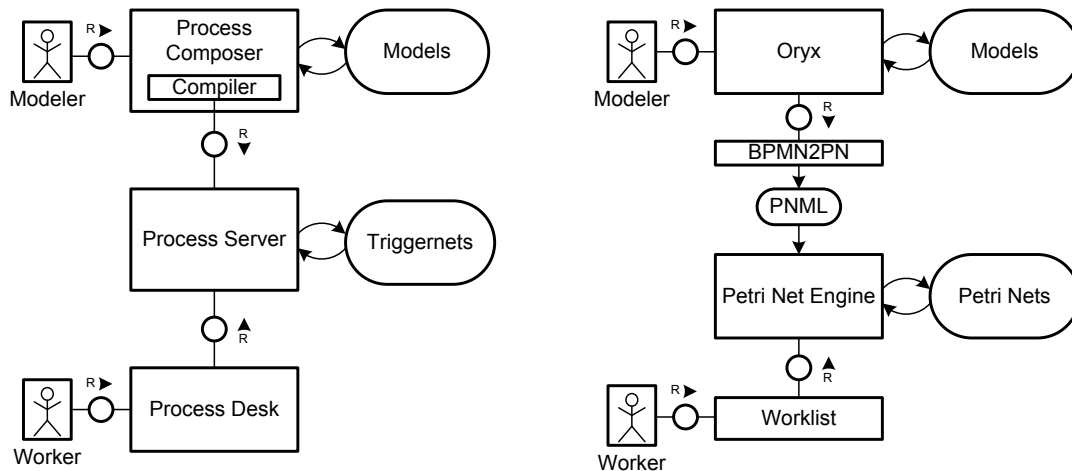


Figure 3.1: Comparing the Architectures of Galaxy (left-hand side) and Oryx Platform (right-hand side) (FMC Structure Diagram)

In order to execute the modeled processes, the platforms have execution environments. In Galaxy processes can be modeled in BPMN. The models are compiled into triggernets and deployed on the Process Server. Triggernets consist of a state and a set of rules describing under which conditions the state of the net changes. The Process Server holds triggernets and their running instances' data. It is responsible for changing the instances' states and triggering the resulting actions by using a rete algorithm. It also controls who can access the process resources. Applications can communicate with the Process Server using adapters.

Oryx has an execution environment for modeled processes, too. Unlike the Galaxy Process Engine, the Oryx engine executes processes as colored Petri nets [Jen96]. Thus, processes have to be modeled as or transformed into Petri nets. At the moment, the Oryx platform offers a transformation from BPMN into Petri nets, only. This is done using a Java application that produces PNML [BCvH<sup>+</sup>03]. Just as the Galaxy server, the Petri net engine holds deployed processes and their instances represented by tokens located on the nets' places.

Galaxy's third important component is the Process Desk, the end user interface to the workflow management system. It is integrated into the user's portal page within his/her company and contains a worklist showing which task the user has to work on. The user can select, allocate and execute those tasks. The execution of a task is done using WebDynpro forms or external applications that are integrated.

On Oryx side, there is a user interface as well. Within the bachelor's project, a worklist was developed as a web-based frontend to the Petri net engine. The layout of this worklist is similar to an email-client UI. Using this frontend the user can allocate tasks and execute them within XForms that are delivered by the engine.

A closer look onto Galaxy's platform architecture can be found in [Kle08]. The Oryx platform architecture, its execution components and the worklist are described in [Ger08] in more detail.

## 3.2 Comparing the Modeling Tools

After regarding the complete platforms, this section takes a closer look on the two modeling environments Oryx and Process Composer. Therefore, the tools' features are examined before the technical implementations are compared.

### 3.2.1 Features

Oryx and the Process Composer share many features. Both are business process modeling environments that offer very similar user interfaces, as figure 3.2 shows. The main elements are two canvases on which the diagrams are drawn via drag and drop. Shape repositories offer the available model constructs and a property area can be used to set attributes. Using a toolbar, the user can access modeling functionality in both tools.

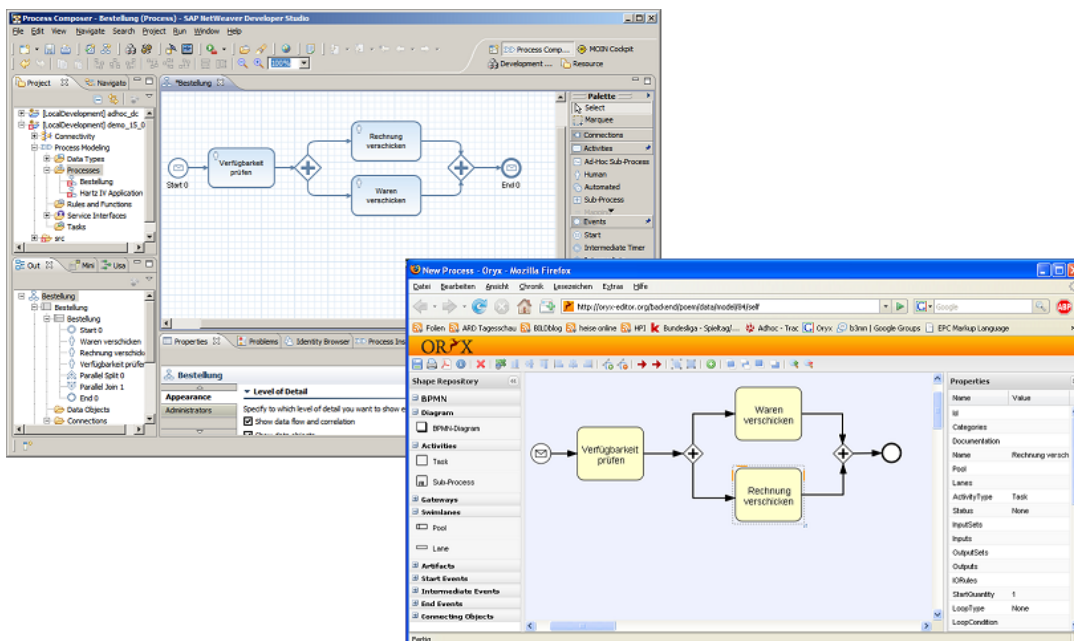


Figure 3.2: Comparing the User Interfaces of Process Composer (left-hand side) and Oryx (right-hand side)

The management of different process models is integrated into the Process Composer. Processes can be organized using projects in a navigator. In Oryx, this is realized using an management page from which the user can choose which process model he/she wants to edit.

That leads to one of the main differences between both tools: The Process Composer is a rich client application that comes with the Galaxy suite and needs to be setup on each computer it should be used with. Oryx, on the other hand, is a web-based solution running within a browser and can therefore be used on every PC with an Internet connection. Due to that, Oryx needs to save its models on a server.

The Process Composer also offers to share process models with other users using a SAP backend. Unlike Oryx, it can permanently store models on the client side, too. However, the leading use case of the Process Composer is to design processes that will be executed in a production systems using the Galaxy engine while Oryx aims for collaborative process modeling.

Therefore, Oryx was designed to be extended easily with new modeling functionality as well as notations. Besides BPMN, it supports several different process modeling languages like EPC, Petri nets and Workflownets. The Process Composer, on the other hand, is optimized for BPMN as only modeling notation, since only the execution of a subset of BPMN constructs is supported by Galaxy.

### 3.2.2 Architectures

In order to compare the architectures of Oryx and Process Composer, first of all both systems' implementations need be introduced.

Figure 3.3 shows Oryx' architecture. As already mentioned, Oryx is a modeling tool that is loaded into a browser. Thereby, it consists of a document describing the process that is currently edited and a number of JavaScript routines offering the modeling functionality.

As one can see, there are two types of JavaScript routines that build up Oryx. First, there is a core that is responsible for creating the canvas, creating an object model and providing a frame that enables plugins to be loaded. These plugins are used to realize basic modeling functionality such as alignments or deletion. But also UI elements such as the property pane or the plugin toolbar itself are realized as plugins [Tsc07]. By the usage of plugins, new functionality can be added easily.

The document which describes the current process model is an XHTML file that contains eRDF. eRDF can be used to write meta data into HTML and can be extracted easily into standard RDF [Czu07]. The backend saves process models in a database and provides functionality to transform them, e.g. into PDF.

The Oryx UI is build using widgets from ExtJS 2.0<sup>1</sup>. Since this is pure JavaScript, it can be displayed in every modern web browser. The shapes that are drawn on

---

<sup>1</sup>[www.extjs.com](http://www.extjs.com)

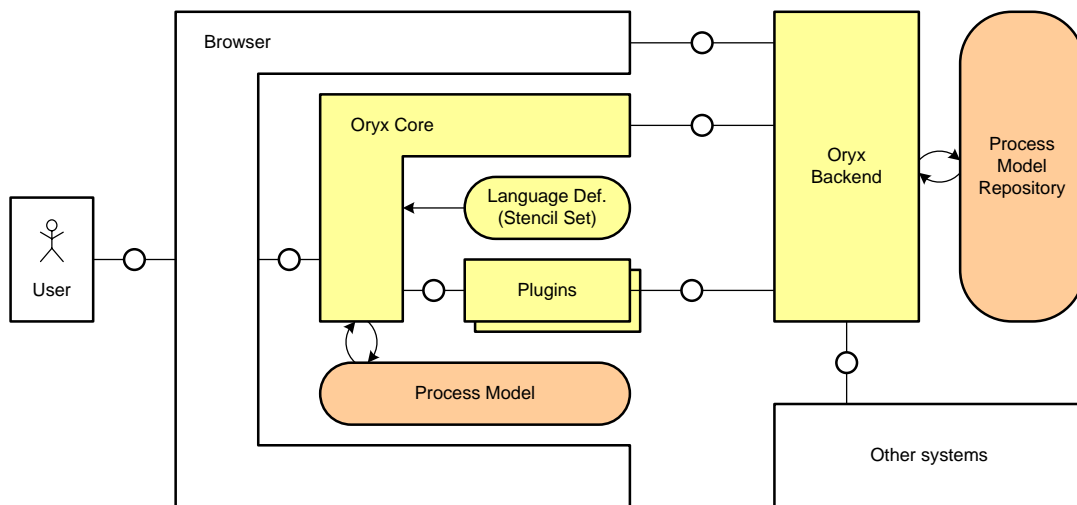


Figure 3.3: The Oryx Architecture (FMC structure diagram)

the canvas are described in SVG. That leads to compatibility problems, since SVG implementations differ from between browsers. Oryx was optimized for running in Mozilla Firefox.

The most interesting part of Oryx are the stencil sets. They are used to describe a modeling notation by listing its element types and rules how elements of those types can be connected or contained. Thereby, each type has attributes and a graphical representation. Stencil sets are written as JSON files and they are loaded completely generically into the Oryx. There are already a number of stencil sets for BPMN, EPC, Petri nets, workflow nets and other notations.

In figure 3.4 the Process Composer's architecture is visualized. The Process Composer is integrated into SAP NetWeaver Development Studio, a SWT-based application.

There are several user interface technologies used in order to provide the user with modeling functionality. First of all, SAP's *Graphics Framework* (GFW [BGHW07]) is used to display the modeling canvas, its shape repository and the property sheets. Tasks can be modeled using SWT form editors. Standard SWT widgets, such as the Outline, the Problems View or the Project Navigator, are used to display process information. Using a NetWeaver XSD-Editor, data-schemas can be created.

The Composer uses SAP's *Modeling Infrastructure* (MOIN) as foundation for its model and meta-model management. A meta-model is used to describe how models of a certain notation may look like. MOIN meta models are described using UML. Both, the models and their meta models are accessed using the MOIN connection agent.

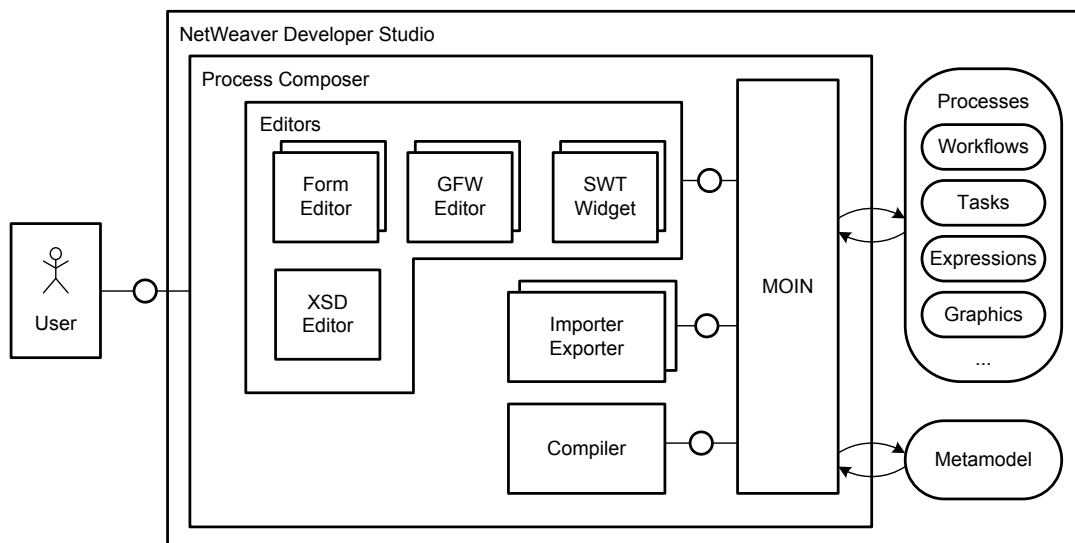


Figure 3.4: The Composer Architecture (FMC structure diagram)

Moreover, the Process Composer provides a trigger net compiler and several im- and exporters as hardcoded components. In Oryx, those must be implemented as plugins since they are notation dependent.

### Usage of Stencil Set vs. Usage of Meta Model

Both, Oryx and Process Composer use a meta model description, but with different motives: Oryx uses stencil sets that are loaded generically into the modeling environment. Only the stencil set needs to be changed in order to change the entries that are shown in the shape repository or the property pane. Connection and containment rules that are specified in the stencil set are checked during modeling but not when the process is saved within the backend.

On the other hand, the Composer's meta model is used to describe how a model must look like. Only correct structures may be saved using MOIN. Further restrictions (e.g. attributes or relations that have to be set) are validated when saving a model and produce visualized errors in the model.

The meta model is not used for a generic creation of modeling UIs or functions. Therefore, a programmer needs to add a new constructs in many places: The meta model must be extended with a new class and the shape repository needs a new registration including hardcoded graphical representation. For changing properties during design-time, every property needs to have an accessing widgets in a property sheet. Like the registration mentioned before, these widgets have to be hardcoded for every property. Especially at this point, data-type dependent widget- or property-sheet-factories would be helpful.

By using OCL rules within the meta model, restrictions are more easy to define in the Process Composer than in Oryx. There, simple containment and connection rules

can be expressed easily, but if a restriction depends not only on an element's types but also on the element's properties, complex and run-time expensive JavaScript methods need to be defined.

### Modeling Data and Functions

As already mentioned, the Process Composer provides the modeler with special editors for data types using XML schema and for functions using an expression language or Java Beans. Data types can be used to model data flow mappings that can be executed at run time. Using functions and data-types, at run time checkable expressions can be build.

None of those concepts can be found in Oryx, yet. Expressions attached to the outgoing arcs of BPMN XOR Gateways, e.g., are defined as strings. Therefore, a simple textbox is used to edit it within the property pane. The same applies to the type of BPMN Data Objects. During our bachelor's project a simple XSD notation for data models was introduced [Kog08].

In future developments, special data types should be defined in the stencil set specification and special widgets should be integrated to edit them. E.g., an attribute type "*XSD-String*" could be introduced and a generic data type editor that opens, when such an attribute is edited could be developed. Within this editor the XSD should be created more easily and other features, such as a validity check, could be integrated.

Since several modeling notations and execution environments could be used with Oryx, there is no chance to offer reusable function modeling as long as there is no central rule engine.

## 3.3 Implementation of Completion Conditions

During our bachelor's project, we extended both environments with the possibility to model Ad-Hoc Subprocesses, as it is shown in figure 3.5. [Kle08] explains, how this has been done within the Process Composer using meta-model changes and UI programming. Within Oryx, only minor changes were necessary, as section 3.4 will show.

But before, this section compares how both tools could be extended with a special aspect: the completion condition. Assuming that Ad-Hoc Subprocesses without these conditions have already been implemented, the following two subsections explain how they can be integrated into the Process Composer (3.3.1) and Oryx (3.3.2).

Section 2.4 already introduced the basic requirements for modeling completion conditions:

- Modeled data as well as execution states should be checked by completions conditions.

- The modeled conditions should be executable. Otherwise, a free-text string in a simple graphical annotation could be used, instead.
- Therefore, the user should have a UI that supports him modeling the condition instead of a textbox he can only type into.

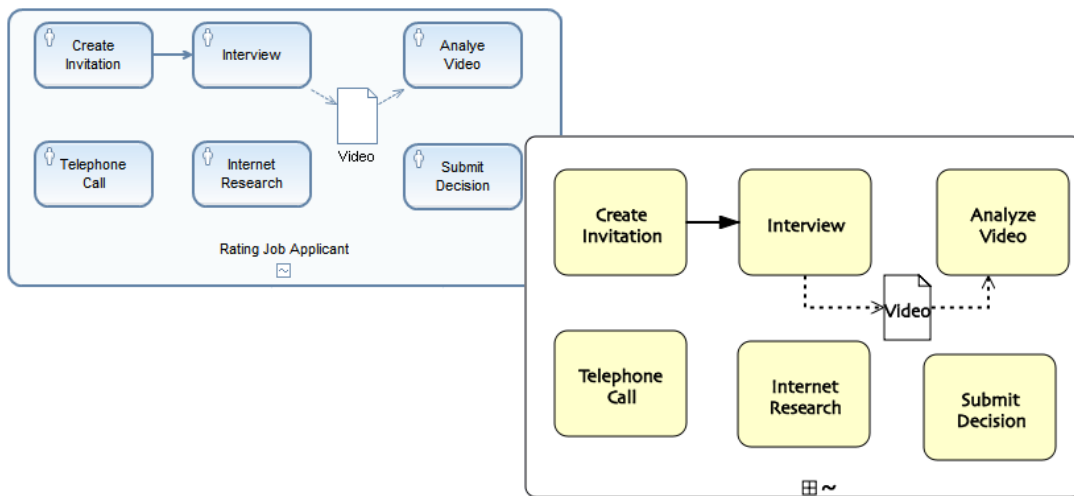


Figure 3.5: Modeling Ad-Hoc Subprocesses with Process Composer (left-hand side) and Oryx (right-hand side)

### 3.3.1 Process Composer

Extending the Composer’s modeling functionality starts with extending the meta model. Assuming, that an Ad-Hoc Subprocess has already been defined, an attribute *“completion condition”* needs to be added. Since expressions are given by the Composer meta model, they can be used here.

An expression is an instance of a function that knows concrete data objects instead of data types. Therefore, a completion condition can be created using functions and data objects that are modeled with the Composer. These given functionalities already solve the requirement of modeling executable conditions with data objects in the scope of Ad-Hoc Subprocesses.

Moreover, the Process Composer has an expression editor that can be used, e.g. to model conditions for the outgoing arcs of a XOR Gateway. So, after creating a new attribute in the meta model, a new property section can be registered as SWT plugin. The section used for the attributes of an Ad-Hoc Subprocess is called **Ad-HocPropertySection** and is located in the package *com.sap.glx.ide.flow.properties*.

Within this section, widgets can be composed. In order to use the Composer expression editor a textfield and a button are placed. The button’s click event can

be used to open the expression editor's dialog while the textfield shows the current condition.

Figure 3.6 shows the expression editor modeling the completion condition from the example (cf. section 2.2). On the right-hand side, the user can choose from a repository of functions as well as of available data objects. Via double-clicks, elements can be added to the textbox on the left-hand side that contains the actual expression.

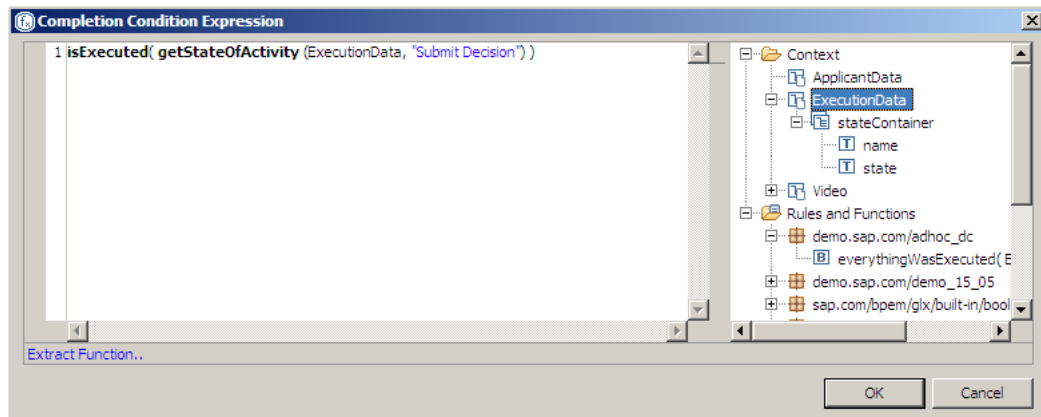


Figure 3.6: Screenshot showing how to model a Completion Condition in Process Composer

Besides data objects that were modeled in a process diagram, execution states should be used within the completion condition, too. Therefore, these meta data need to be represented by data objects that are available at design- and at run-time. The implemented execution-state data-objects are part of their parent subprocesses but, since they are meta data, have no graphical representation in the diagram.

As one Ad-Hoc Subprocess may have many subtasks and the representing meta data objects would be shown in every expression editor, we decided to represent all execution states by a single data object per subprocess. Due to the fact that data types must be described using static XSD, we decided to use a collection of  $n$  state entries that map from a sub-activity's name to its execution state.

Besides this data-type, some standard-functions need to be defined. E.g., a function that checks if all contained activities have been executed or functions that get entries out of the map and check their states. In the current Composer implementation, these functions only exist as signatures with no Java Bean implementation. However, the following code snippet shows function implementations that are relevant for the example above:

```

public static ExecutionState getStateOfActivity(
    ExecutionContainer[] list, String name){
    for (ExecutionContainer entry : list) {
        if (entry.activity.equals(name){
            return entry.state;
        }
    }
    return null;
}

public static boolean isExecuted(ExecutionState state){
    // ExecutionState inherits from String!
    return ( state != null && state.equals("executed") );
}

```

Now, it is obvious that the completion condition in figure 3.6 evaluates to true if and only if *"Submit Decision"* has been completed.

### 3.3.2 Oryx

Unlike the Process Composer, Oryx has no concept for functions or expressions either at design- nor at run-time. Therefore, the implementation of a modeling tool is less restricted by existing solutions. Execution states, e.g., do not have to be represented by data objects but can be integrated directly. Since no UI for the creation is given, the completion condition dialog's layout can be designed to be more intuitive.

The following explanations assume that the used BPMN stencil set already knows subprocesses with an *isAdHoc* and an *AdHocCompletionCondition* attribute. Since no functions or expressions are known, the completion condition is a simple string that can be edited using a standard textbox within the property pane. Therefore, a more supporting UI is needed.

The first step for creating an editor customized for a certain property is to register a new plugin within the **plugins.xml** file. The here introduced plugin is named **adHocCC.js** and should only be loaded when an BPMN stencil set is used. This can be done by setting a required stencil set namespace.

Within the plugin, a dialog is built using the ExtJS 2.0 framework. If an Ad-Hoc Subprocess is selected in the canvas, the dialog can be opened by clicking on a toolbar button. Figure 3.7 shows the dialog that can be used to create expressions based on either execution states or modeled data. Moreover, logical operators can be used to combine those sub-expressions to a single one.

To model an execution state expression, an activity can be chosen from the first combobox. Additionally, a state that should be reached in order to evaluate to true,

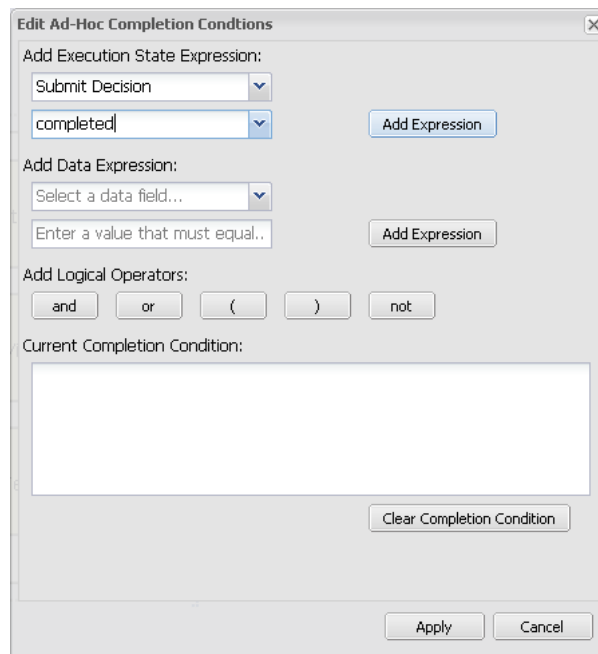


Figure 3.7: Screenshot showing how to model a Completion Condition in Oryx

can be chosen from the second combobox. In the screenshot, *"Submit Decision"* and *"completed"* have been selected. By clicking on the *Add Expression* button, the following state-based expression will be added to the textbox below:

```
stateExpression('resource1', 'completed')
```

Thereby, *resource1* is the ID of the activity *"Submit Decision"*.

Data-based expression can be built using the third combobox and the textfield below. The combobox contains all available data object's fields. At run time, the field should be compared with the selected data field. An example for a data-based expression is:

```
dataExpression('dataObjectResourceID', 'fieldName', 'value')
```

After editing the completion condition, the new value is written into Oryx' data structure. The translation of expressions modeled with this dialog into executable guard conditions will be explained in chapter 4.

## 3.4 Survey of Changes in Oryx

The following list gives a short overview on all changes that were performed in order to support modeling BPMN Ad-Hoc Subprocesses with Oryx:

- Oryx' BPMN stencil set [Pol07] did already support the BPMN specification almost completely, when our bachelor's project started. In order to extend

Oryx with BPMN that could be transformed into executable Petri nets, we **created a copy of this stencil set and extended it** with additional constructs. The new stencil set is described by the **bpmnexecutable.json** file in Oryx' SVN repository.

- The attributes **AdHocOrdering** and **AdHocCompletionCondition** **needed to be added** to this stencil set's *Subprocess*. However, these attribute were defined for *Pools*, which was reverted.
- Due to used serialization technology, the identifiers of all attributes within a stencil set need to be lower case, only. Otherwise, instances' properties cannot be saved and loaded again. Nevertheless, a lot of attribute identifiers contained upper cases characters. This **bug was fixed** for the whole stencil set (including the *isAdHoc* attribute of Subprocesses).

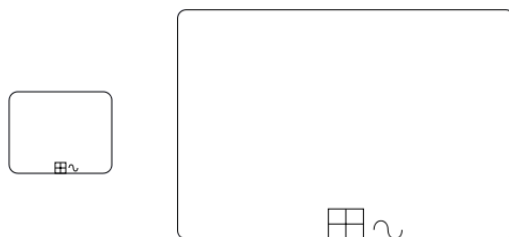


Figure 3.8: Subprocesses' Scaling Problem

- Scaling subprocesses to a convenient size leads to the problem of oversized and unproportional markers, as shown in figure 3.8 This has been fixed by **disabling the resizable attributes of markers and setting new anchors as well as transformations** within the subprocesses' SVG file together with members of the Oryx team.
- In order to model completion conditions a new plugin called **adHocCC.js** **was written** (cf. section 3.3.2)

As one can see, besides the completion condition modeling plugin presented in 3.3.2, only minor changes needed to be performed in order to model Ad-Hoc Subprocesses conveniently. In order to execute these task in the given Petri net engine, more development was required.

## 4. Transformation

After modeling a process with Oryx or the Process Composer, it is desired to execute it. Therefore, the model needs to be transformed into an executable representation. This chapter shows how Ad-Hoc Subprocesses are transformed into Petri nets that can be executed by the Oryx engine.

Therefore, section 4.1 introduces the general transformation of BPMN *Activities* into Petri nets under Oryx. In section 4.2, the translation of *Ad-Hoc Subprocesses* into Petri nets including the conversion of completion conditions is explained. Afterwards, the transformation of dependency constructs is shown in section 4.3.

### 4.1 Transforming BPMN Activities into Petri nets

As already mentioned in section 3.1, the Oryx platform contains a workflow engine that executes colored Petri nets. These nets consist of places and transitions in a bipartite structure. On places, tokens are located that identify the state of the net. Each place has a data schema that must be fulfilled by the contained tokens.

A transition is enabled if all its input places are filled and the attached guard condition evaluates to true. Firing a transition removes all tokens on the input places and set transformed tokens onto the output places. This transformation can either be done automatically or manually.

The following two subsections present how BPMN diagrams are transformed into Petri nets. Therefore, subsection 4.1.1 presents a concept for mapping BPMN Activities onto Petri nets and subsection 4.1.2 shows how this concept is technically realized in Oryx.

### 4.1.1 General Activity Lifecycle

Firing a Petri net transition can be seen as a conceptually instantaneous action while working with a BPMN Activity includes several steps. First, a ready activity must be allocated before the actual form can be edited. During this process, the activity may be suspended and resumed before the results can be submitted.

Figure 4.1 shows the lifecycle of an activity the engine should execute. This also includes the possibility to skip a ready activity. Further states that are needed to realize delegation functionality [Nag08, Mas08] are not described in this thesis.

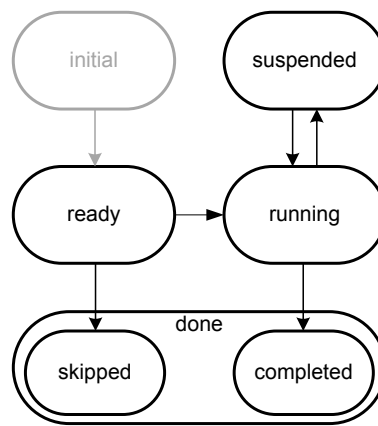


Figure 4.1: State Transition Diagram of a BPMN Activity

Since all these states cannot be represented by a single instantaneous transition with only one input- and one output-place, we decided to represent the BPMN Activity as a whole Petri net with one input and one output-place. Thereby, the states in diagram 4.1 are mapped onto Petri net places while transitions are mapped onto Petri net transitions. Figure 4.2 shows the resulting Petri net.

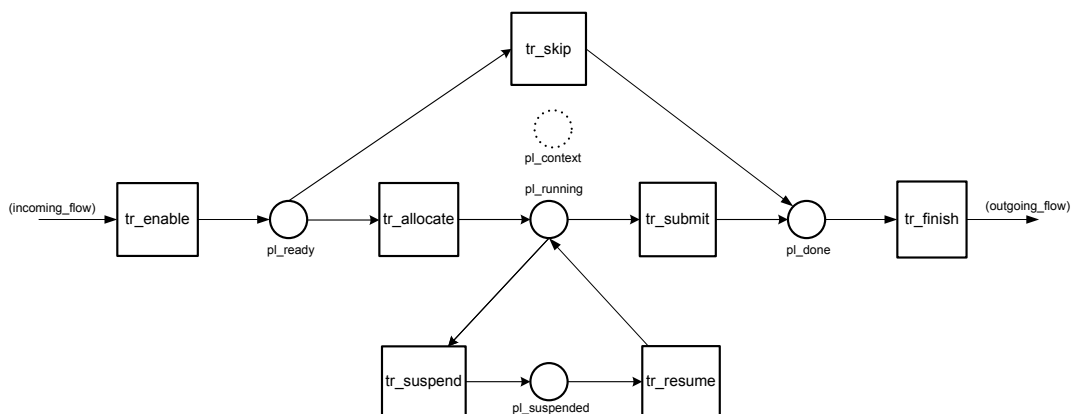


Figure 4.2: BPMN Activity represented as Petri Net

Note, that there is no place representing the initial-state introduced in figure 4.1, since all tasks are initialized when a process instance is spawned. As the finish transition fires immediately, the two "done" states are represented by one place only. Moreover, all transitions read data from and write data to the context place. Due to understandability, the responsible arcs have not been shown.

The task's context place stores meta data, such as the current owner of the task, timestamps and its current state. Although this is a redundant information that is also represented by the other places, it is much more easy to access on the context place, since only one read-arc is necessary. Within the context place the two done states "skipped" and "completed" are distinguished.

The presented transformation is very similar to the transformation that is used within YAWL<sup>1</sup>, as described in [Wes07, chapter 4.5]. There, the task states *enabled*, *exec* and *completed* are represented in a Petri-net like structure. These states can be mapped to the places *ready*, *running* and *done* in figure 4.2. Suspending or skipping activities is not provided in YAWL.

### 4.1.2 Transformer Implementation

In Oryx, the transformation can be triggered using the plugin `pnmlexport.js` that is represented in the toolbar. This plugin sends the current process model as RDF-Document to a Java servlet that carries the actual transformation algorithm.

First, the transformation deserializes the RDF into a BPMN object structure using a BPMN meta model built by Java classes. The BPMN model is translated into a Petri net model. Again, Java classes are used as Petri net meta model. Finally, the Petri net is serialized as PNML and can be deployed into the Petri net engine.

The crucial part of the transformation is the conversion between BPMN model and Petri net model. Here, different converters have been implemented. The transformations described in section 4.1.1 that results in an executable BPMN representation are performed using the class `ExecConverter`.

Further information about the transformation of standard BPMN into Petri nets can be found in [Mas08].

## 4.2 Transforming Ad-Hoc Subprocesses into Petri nets

In section 2.3, Ad-Hoc Subprocesses have been introduced. These subprocesses may contain un-ordered BPMN Tasks, have an `AdHocOrdering` and an `AdHocCompletionCondition` attribute. While the ordering attribute defines whether subtasks can be executed in parallel, the completion condition determines when the subprocess' execution finishes.

---

<sup>1</sup><http://www.yawl-system.com/>

In the following two subsections, the transformation of these Ad-Hoc Subprocesses is explained. Thereby, subsection 4.2.1 presents the structural transformation while subsection 4.2.2 focuses on converting the modeled completion condition.

### 4.2.1 Structural Transformation

Before regarding the transformation of Ad-Hoc Subprocesses, the conversion of standard BPMN *Subprocesses* should be examined. The already mentioned execution semantics of YAWL uses subprocess instances with the same interface as task instances. These subprocesses simply activate contained subtasks when they are started and finish their execution when the contained process has completed.

Transferring this to the world of a BPMN-to-Petri-net mapping, a subprocess needs to have one input and one output. A start transition should activate the first contained subtask, while an end transition waits for the last subtask to complete. Since YAWL only delivers execution semantics for ordered subprocesses, additional constructs have to be created that enable the execution of transformed Ad-Hoc Subprocesses.

Like a normal subprocess, an Ad-Hoc Subprocess needs an input place and an output place in order to be connected with the surrounding process. As the contained subtasks are not ordered, the start place leads to the start transition that puts tokens on the start places of all contained subtasks. The end place of every subtask is connected with the subprocess' standard end transition that leads to the end place.

Figure 4.3 shows this structure as a Petri net. At the top, transitions and places are shown that occur only once per subprocess. At the bottom, the transformed structure of every contained subtask is drawn and connected with the subprocess-specific elements. The two transformed subtasks and the dot groups in between express that there are many subtasks within a transformed Ad-Hoc Subprocess.

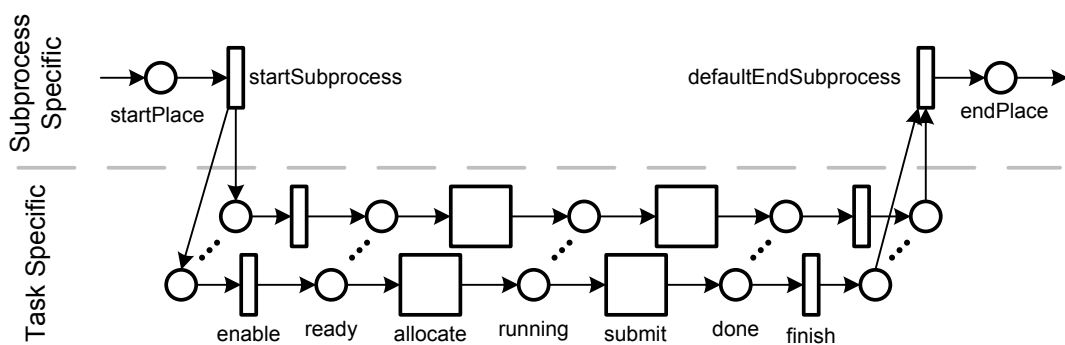


Figure 4.3: First Ideas on Transforming a BPMN Ad-Hoc Subprocess into a Petri Net

In order to distinguish between automatically and manually fired transitions, automatic transitions have a thin shape while manual transitions have square ones.

Note, that places and transitions that are responsible for suspending, resuming and skipping the subtasks are not shown due to understandability.

The Petri net in figure 4.3 already realizes the transformation of a parallel ordered Ad-Hoc Subprocess that has no completion condition and, therefore, will be finished, when all subtasks have been executed or skipped.

In order to realize a completion condition check in an Ad-Hoc Subprocess, some additional constructs are needed. Figure 4.4 introduces the already known Petri net with those extensions. Thereby, only one contained subtask is shown, for the sake of brevity. Newly introduced parts are drawn in black while already known parts are grayed.

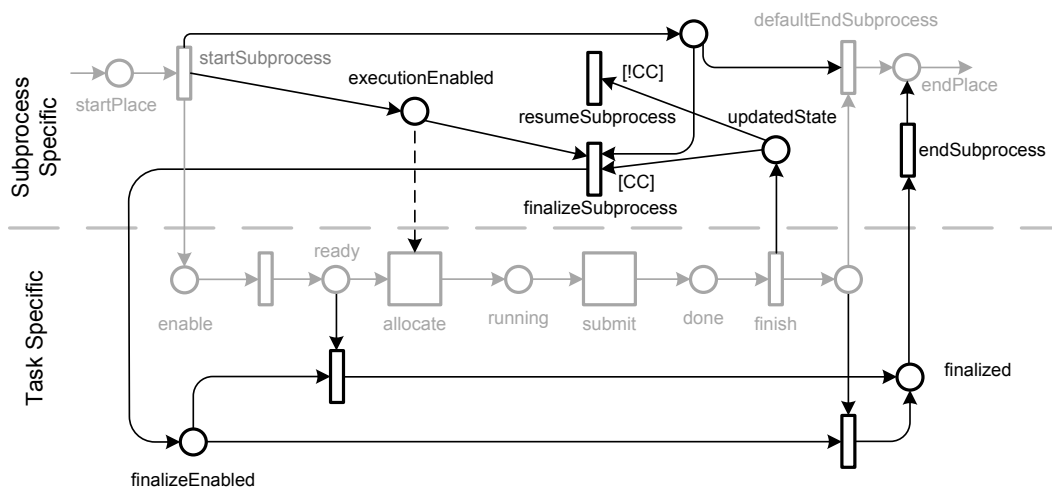


Figure 4.4: Transforming a Parallel Ordered BPMN Ad-Hoc Subprocess

First of all, the *updatedState* place is relevant: As explained in section 2.4, the completion condition should be checked, whenever a data object or execution state changes. This only happens when a subtasks is either skipped or completed and, therefore, new data is written to data places. So, when a subtask finishes its execution, it puts a token onto the *updatedState* place. That enables either the *resumeSubprocess* or the *finalizeSubprocess* transition depending on the disjoint completion condition guard conditions.

If the completion condition evaluates to false, the *resumeSubprocess* transition simply removes the token from the *updatedState* place. Otherwise, by firing the *finalizeSubprocess* transition, the finalization of the ad-hoc subprocess is triggered.

Each subtask is finalized by either removing a token from its ready place or its finished place. Thereby, running tasks will not be skipped. The finalization of the Ad-Hoc Subprocess waits for those tasks to complete. After all subtasks have been

finalized, the *endSubprocess* transition that is different from the *defaultEndSubprocess* transition can fire and the subprocess completes its execution.

Moreover, the figure shows that the allocation of new subtasks is blocked after the completion condition evaluated to true. This is done using the *executionEnabled* place that is filled when the subprocess is started. Via dotted read-arcs, subtasks' allocate transitions can only fire if a token is located on the *executionEnabled* place. In order to fire *finalizeSubprocess*, this token needs to be consumed.

The place that is filled by *startSubprocess* and can be cleared by either *finalizeSubprocess* or *defaultEndSubprocess* eliminates the possibility of a critical race: When the last subtasks finishes its execution and, therefore, the completion condition evaluates to true, both mentioned transitions are enabled. Without the extra place, by firing *finalizeSubprocess* before *defaultEndSubprocess*, tokens would remain within subtasks' finalization places.

Besides the missing transformation of modeled completion condition that will be regarded in subsection 4.2.2, only one issue is not shown in figure 4.4:

The finalize transition that removes a token from a subtask's ready place needs to record this procedure in the particular context place's token. Therefore, a read arc and a write arc are needed as well as a XSLT transformation that writes new timestamps and "skipped" as new state into the token.

The solution presented so far solves the problem of transforming Ad-Hoc Subprocesses into Petri nets only for parallel ordered subprocesses. Thereby, an implementation was chosen that lets running subtasks complete their execution when the completion condition evaluates to true. Though, it does not regard the possibility that one of those task might change the completion condition's evaluation to false again. Due to limited time, a second solution that would re-check the condition every time a still running task finishes its execution has not been implemented.

In order to transform sequentially ordered Ad-Hoc Subprocesses, only a few changes are required, as figure 4.5 shows. Instead of the *executionEnabled* place, a *synchronize* place is used. The allocation of a subtask removes this place's token which blocks other subtasks to be allocated, too. If the completion condition evaluates to false, a new token is set onto the *synchronize* place. Again, only one contained subtask is shown and already known parts are grayed within the figure.

All presented solutions have been implemented in the Java transformer. Most of the implementation is located in the method **ExecConverter.handleAdHocSubprocess()** that creates the Petri net structures and in the class **ExecConversionContext** that holds a subprocess-children map.

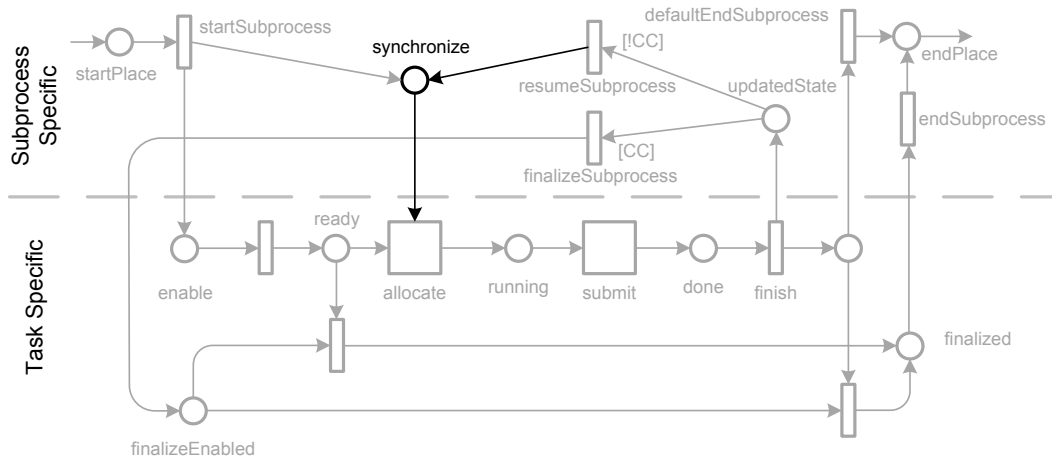


Figure 4.5: Transforming a Sequentially Ordered BPMN Ad-Hoc Subprocess

## 4.2.2 Transformation of Completion Conditions

Using the Oryx Petri net engine, completion conditions can be realized using guard expressions at *resumeSubprocess* and *finalizeSubprocess*. Therefore, the modeled completion condition needs to be translated into an evaluable expression.

Within completion conditions, data objects can be checked as well as execution states. In the engine, BPMN Data Objects are represented using special data places while execution states of contained subtasks can be accessed on their context places. Places define a data schema using *locators*. In order to access a certain field in a token, the particular transition needs to be connected with the token's place.

Figure 4.6 shows how *resumeSubprocess* and *finalizeSubprocess* access the relevant data using read-arcs. The disjoint expressions  $CC$  and  $!CC$  represent the translated completion condition. As in figure 4.3, the dot groups indicate that there may be many data and context places that can be accessed.

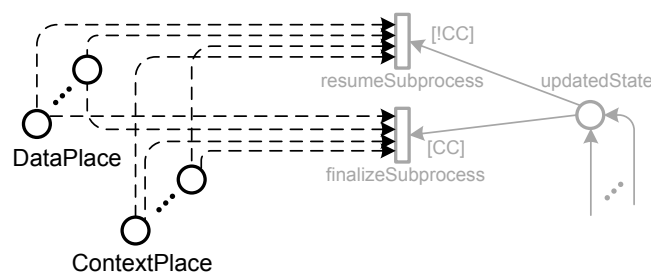


Figure 4.6: Completion Condition Checking Transitions

The Ruby<sup>2</sup>-based Petri net engine offers the possibility to evaluate guard expression using the Ruby *eval* function. This function takes a string that is written in Ruby syntax and executes it. Thereby, e.g. logical expressions can be checked.

Section 3.3.2 introduced a plugin that can be used to model completion conditions with Oryx consisting of certain sub-expressions. In table 4.1 all these sub-expressions are mapped to Ruby expressions that can be checked during execution.

| Description     | Modeled Expression               | Ruby Expression            |
|-----------------|----------------------------------|----------------------------|
| Execution State | stateExpression('X1', 'X2')      | pl_context_X1.status=='X2' |
| Data            | dataExpression('Y1', 'Y2', 'Y3') | pl_data_Y1.Y2=='Y3'        |
| Logical And     | &                                | &&                         |
| Logical Or      |                                  |                            |
| Logical Not     | !                                | !                          |
| Logical (       | (                                | (                          |
| Logical )       | )                                | )                          |

Table 4.1: Mapping Modeled Expression onto Ruby Expressions

The usage of an eval function is risky, since dangerous code containing system calls, e.g., could be injected and executed. Therefore, the mapping shown above does not transfer unidentified input expressions into the output string. By doing so, the injection does not become impossible, but hindered in most cases.

In cases of incorrect completion conditions, the resulting Ruby expression might be not-evaluatable at run time. This should be reported to the modeler that started the transformation. However, in cases of no or a not-evaluatable completion condition, the standard completion condition realized by the *defaultEndSubprocess* transition works as a safeguard.

### 4.3 Transforming Dependency Constructs into Petri nets

In section 2.3, concepts were introduced that can be used within an Ad-Hoc Subprocess to express dependencies between contained subtasks. Thereby, two easily understandable constructs were chosen: Data dependencies and sequence flow.

The idea of data dependencies comes from the concept of *Case Handling*. In our semantics, they express that a data object must be filled in order to start a depending activity. Since such a start condition does not always logically depend on a certain data object and BPMN modelers are used to model control flow by sequence flow instead of data flow, sequence flow is also needed.

<sup>2</sup><http://www.ruby-lang.org/>

The semantic of sequence flow within an Ad-Hoc Subprocess is similar to the data dependencies' semantic: If a sequence flow goes from A to B, B can only be started after A has finished. B does not have to start immediately or at all.

In order to implement this kind of sequence flow, a place between A and B can be used. A's *finish* transition puts a token onto this place while B's *enable* transition consumes it. Now, it is possible, that a subtask is not ready when the completion condition evaluates to true. Therefore, a third finalize transition must be integrated. This transition removes the token the *startSubprocess* transition has placed in order to enable all its subtasks. Figure 4.7 shows a cutout of the newly integrated constructs.

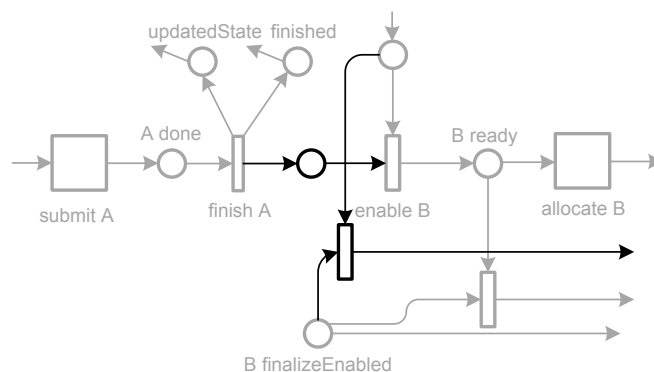


Figure 4.7: Integration of Sequence Flow in Transformed Ad-Hoc Subprocess (Cutout)

In order to add data dependencies, a further extension has to be made. Since the content of a data object may be filled and emptied again, it is not wise, to let the automatically fired enable transition depend on the data place. It may take time, until the subtask will actually be allocated and by then, the data dependency might evaluate to false again.

Therefore, a data dependency is realized using a read-arc from the data place to the allocate transition and a corresponding guard condition. This is shown in figure 4.8.

The *filled-guard-condition* is realized using the assumption, that a data object is filled when all its fields carry not empty values. E.g., the data object *"Video"* from the scenario in section 2.2 carries a name and an URL pointing to the actual video file. Both attributes are represented as strings. The guard condition of the depending task *"Analyze Video"* is:

```
pl_data_resourceX.name != "" && pl_data_resourceX.url != ""
```

In future development, further data dependency conditions should be integrated that can be defined by a modeler. Therefore, a similar UI as shown in 3.3.2 could

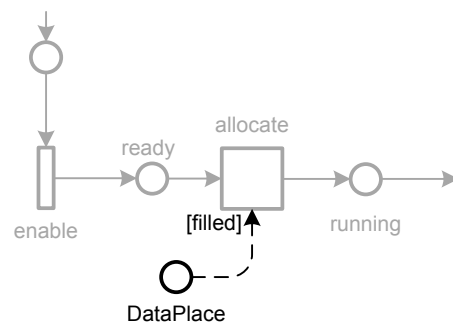


Figure 4.8: Integration of Data Dependencies in Transformed Ad-Hoc Subprocess (Cutout)

be used. Analog to the mapping of data expressions used in completion conditions, these conditions could be transformed into Ruby expressions.

# 5. Conclusion

In this final chapter, the performed work both in the context of design-time-based adhocness and the whole bachelor's project is reflected.

## 5.1 Ad-Hoc Subprocesses

Within this thesis, I presented modeling constructs that can be used to model flexible run-time behavior at design-time. The *Ad-Hoc Subprocess* that was already motivated and specified in [Kle08] was taken up and its completion condition was examined in detail.

We implemented the Ad-Hoc Subprocess in both, the Galaxy Process Composer and in Oryx. Thereby, the different concepts of both tools, especially at the usage of meta models had to be regarded. While Oryx uses stencil sets as generic description of modeling constructs and properties, the Process Composer uses one BPMN meta model to validate process models.

Besides graphical representation of the subprocess, properties such as the completion condition have to be editable. Therefore, two editors have been integrated and developed. Since the complex Galaxy Process Server is still under development and our project went only for a short period of time, execution semantics has only been implemented within Oryx.

Here, BPMN can be transformed into executable Petri nets. This transformation was extended with the possibility to convert Ad-Hoc Subprocesses. Besides the basic structure, also the transformation of completion conditions and dependency-constructs had to be regarded.

Due to the fact that we implemented prototypes, there remain open issues: The standard functions that are used to specify a completion condition within the Process Composer only exist as signatures but not as deployed JavaBeans. The conversion

of parallel ordered Ad-Hoc Subprocesses does not react properly when a completion condition evaluates to false, after it evaluated to true. Moreover, the evaluation of guard conditions using Ruby's eval-function may lead into security problems and only simple data dependencies are supported, yet.

Additionally, the transformation of Ad-Hoc Subprocesses only works within Oryx. For SAP AG, a conversion of the presented transformation concept into the world of Galaxy, where triggernets are used, is an important issue.

## 5.2 Bachelor's Project

During the bachelor's project, many different problems were examined and many potential solutions researched. Finding relevant problems as well as suitable and realizable solutions was one of the major tasks that cannot be reflected by the resulting theses.

These were written in order to describe the outcoming results Delegation, Re-evaluation and Design-Time-Based Adhocness. Although, a lot of time was used for identification and examination of concepts that were discarded later, this was also valuable work, since it helped to identify the really important issues.

Technical problems occurred during the development of Galaxy code. For certain activities, such as building a new meta model or even running the Process Server, a VPN connection with SAP's intranet is required. Since we only had one SAP computer that is capable to connect this net, work was often difficult. Additionally, there was no code synchronization with Galaxy's running development for 4 month.

By implementing modeling and execution prototypes, the found solutions have been demonstrated. At the HPI, the prototypes can be used for further research. Additionally created components, such as the security system [Nag08] and the worklist [Ger08] can be used in future Oryx releases.

On Galaxy side, a first list of use cases and requirements that should be supported and met in order to provide a more flexible workflow management system exist now. Moreover, there are concepts for solving these relevant issues. The realizability of Delegation, Re-evaluation and Ad-Hoc Subprocesses is demonstrated by running prototypes.

# Bibliography

- [AtHEvdA06] Michael Adams, Arthur H. M. ter Hofstede, David Edmond, and Wil M. P. van der Aalst. Implementing Dynamic Flexibility in Workflows using Worklets. In *BPMCenter Report BPM-06-06*. BPMcenter.org, 2006.
- [BCvH<sup>+</sup>03] Jonathan Billington, Søren Christensen, Kees M. van Hee, Ekkart Kindler, Olaf Kummer, Laure Petrucci, Reinier Post, Christian Stehno, and Michael Weber. The Petri Net Markup Language: Concepts, Technology, and Tools. In *24th International Conference on the Applications and Theory of Petri Nets (ICATPN)*, LNCS, pages 483–505, Eindhoven, The Netherlands, June 2003.
- [BGHW07] Christian Brand, Matthias Gorning, Stephan Heik, and Stefan Werner. SAP graphics framework - an introduction. Technical report (SAP confidential), SAP, March 2007.
- [BGK<sup>+</sup>07] Stefan Baeuerle, Ulrike Greiner, Marek Kowalkiewicz, Sonia Lippe, Marita Kruempelmann, and Ruopeng Lu. Enterprise Services Architecture - Workflow@AP. Technical report (SAP confidential), SAP Research, 2007.
- [bpm08] Business Process Modeling Notation, V1.1. Technical report, Object Management Group (OMG), Jan 2008. <http://www.omg.org/spec/BPMN/1.1/PDF/>.
- [Czu07] Martin Czuchra. Oryx - Embedding Business Process Data Into the Web. Bachelor's Thesis, Hasso Plattner Institute at the University of Potsdam, July 2007.
- [DGKW08] Gero Decker, Lutz Gericke, Stefan Krumnow, and Mathias Weske. Prozessmodellierung und -ausführung im Web. Technical report, Hasso-Plattner Institute at the University of Potsdam, Potsdam, 2008.
- [Ger08] Lutz Gericke. Execution Perspective for Ad-Hoc Business Processes. Bachelor's Thesis, Hasso-Plattner Institute at the University of Potsdam, June 2008.

- [Jen96] K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. Volume 1*. Springer, 1996.
- [Kle08] Matthias Kleine. Extending Galaxy Composer with Ad-Hoc Constructs. Bachelor's Thesis, Hasso-Plattner Institute at the University of Potsdam, June 2008.
- [Kog08] Alexander Koglin. Scenarios, Usability and XForms in Ad-Hoc Business Processes. Bachelor's Thesis, Hasso-Plattner Institute at the University of Potsdam, June 2008.
- [Mas08] Philipp Maschke. Execution and Re-evaluation of BPMN Processes. Bachelor's Thesis, Hasso-Plattner Institute at the University of Potsdam, June 2008.
- [Nag08] Mike Nagora. Roles and Delegation in Ad-Hoc Business Processes. Bachelor's Thesis, Hasso-Plattner Institute at the University of Potsdam, June 2008.
- [Pol07] Daniel Polak. Oryx - BPMN Stencil Set Implementation. Bachelor's Thesis, Hasso Plattner Institute at the University of Potsdam, July 2007.
- [PSvdA07] Maja Pesic, Helen Schonenberg, and Wil M. P. van der Aalst. DECLARE: Full Support for Loosely-Structured Processes. In *EDOC*, pages 287–300, 2007.
- [RRvdA03] Hajo A. Reijers, J. H. M. Rigter, and Wil M. P. van der Aalst. The Case Handling Case. *Int. J. Cooperative Inf. Syst.*, 12(3):365–391, 2003.
- [SSO01] Shazia W. Sadiq, Wasim Sadiq, and Maria E. Orłowska. Pockets of Flexibility in Workflow Specification. In *ER '01: Proceedings of the 20th International Conference on Conceptual Modeling*, pages 513–526, London, UK, 2001. Springer-Verlag.
- [Tsc07] Willi Tscheschner. Oryx - Documentation. Bachelor's Thesis, Hasso Plattner Institute at the University of Potsdam, July 2007.
- [vdAWG05] Wil M. P. van der Aalst, Mathias Weske, and Dolf Grünbauer. Case handling: a new paradigm for business process support. *Data Knowl. Eng.*, 53(2):129–162, 2005.
- [Wes07] Mathias Weske. *Business Process Management*. Springer, 2007.