ulm university universität ulm

Ulm University | 89069 Ulm | Germany

Faculty for Engineering and Computer Science
Institute for Databases and Information Systems

Enabling Personalized Business Process Modeling: The Clavii BPM Platform

Master Thesis at Ulm University

Submitted by:
Klaus Kammerer
Klaus.Kammerer@uni-ulm.de

Reviewer:
Prof. Dr. Manfred Reichert
Prof. Dr. Peter Dadam

Supervisor:
Dipl.-Inf. Jens Kolb

2014

Version of August 28, 2014

ACKNOWLEDGEMENTS:

First and foremost I offer my gratitude to my supervisor, Jens Kolb, who has supported me throughout my studies with his enthusiasm and knowledge.
I would like to thank Prof. Dr. Manfred Reichert for making this thesis possible.
Many thanks to my fellow students and Clavii workmates Kevin Andrews, Stefan Büringer, and Britta Meyer.
I would particularly like to thank my family, who made my studies possible and encouraged all my decisions.
Finally, I thank Vanessa Schmitt. She was always cheering me up and stood by me through the good times and bad.

Typeset: PDF-LaTeX 2$_\varepsilon$
Print: Ulm University

Abstract

Increasing adoption of business process management systems has resulted in large business process models comprising hundreds of activities. Particularly, such process models are hard to understand and maintain. This issue requires innovative approaches to simplify and personalize process models. Therefore, this thesis introduces fundamentals for process views offering personalized perspectives for process participants by abstracting not necessary information. Furthermore, an approach for a domain-specific process modeling language, so-called Process Query Language, is presented. The latter offers process modeling notation independent abilities to define, search, and modify process models as well as process views. The proof-of-concept implementation, so-called Clavii BPM platform, shows up as integrated solution for simple, web-based business process modeling and execution. Thus, it implements basic concepts for process views and the PQL language.

Contents

1

Introduction

In times of the information age companies are more than ever confronted with big challenges [19]. On the one hand side customers ask for even lower prices, faster product delivery and higher support—on the other side companies aim at maximizing their return on invest. One solution may be to manage and optimize efficiency through managing internal process flows.

Companies often attach significant importance to *Business Process Management* (BPM) [26], but comprehensive BPM software platforms are costly and require profound BPM knowledge. Additionally, business processes are subject of continuous change demanding for advanced technologies supporting a fast adaption and optimization of company wide business processes. BPM software platforms have to address these challenges.

Large companies are able to use a great variety of BPM platforms, which include *Business Intelligence*[1], Service-Oriented Architectures or *Business Rules*. Small and medium-sized companies are often overcharged with an integrated BPM platform, since BPM knowledge—or available platforms are too complex and expensive for them [34]. BPM platforms are often developed for comprehensive mapping of ideally all accruing internal and external workflows. They follow a top-down approach, which requires a high-level view on a companies processes.

This thesis presents the *Clavii BPM platform* (*Clavii* for short), an integrated, web-based solution to manage, execute, and optimize process models. It illustrates different concepts to simplify process models. Clavii is optimized for small and mid-sized companies with limited experiences with BPM. Therefore, Clavii offers a simple and self-explaining user interface (UI) trying to close the gap between organizational and technical BPM aspects. Assisted process modeling bypasses technical difficulties to ensure, for example, correct control and data flow in process models. Advanced concepts, e.g., *process views* for process model abstractions, streamline the management of large and complex business processes. As a research result for this topic the *Process Query Language (PQL)* is presented. PQL is a domain-specific language for process modeling and enables the definition of notation-independent change and abstraction operations. Moreover, Clavii features a simplified data flow by supporting business objects and allows for an easy automation of BPM tasks. Hence, end-users are not required to develop software component tasks, like premised for other BPM platforms. Therefore, execution-ready generic tasks for process automation can be obtained in a store and require little or no configuration effort. Furthermore, process models for common activities, like delivery processing, can also be obtained in the store to further minimize initial obstacles.

Section 2 introduces fundamentals on business process management. Section 3 presents theoretical approaches for process views. Section 4 introduces PQL as well as a generic software architecture supporting PQL. Section 5 presents the Activiti BPM platform as base of the developed Clavii BPM platform. Section 6 introduces the Clavii BPM platform by describing various innovative principles and its software architecture. Section 7 discusses related work. A conclusion is given in Section 8.

[1] Technologies to transform raw data into meaningful and useful information for business analysis purposes

2

Fundamentals on Business Process Management

In the following fundamentals of Business Process Management, serving basic knowledge for further aspects of this thesis, are introduced [29, 70, 82]. Section 2.1 explains Business Process Management and the BPM lifecycle in general. Section 2.2 describes essentials, like the definition of a process model, its graphical description model and an introduction to BPMN 2.0, a de-facto standard for todays process modeling notations. Executing business processes is explained in Section 2.3.

2.1 Business Process Management

The European Association of BPM (EABPM) defines *business process management* as following: *"Business Process Management or BPM is synonymously used for process management. A process is a set of defined operations (i.e., activities, tasks), that are executed by human or a system to achieve a goal. [...] Business process management is a systematical approach to capture, execute, document, measure, monitor and control automated and non-automated processes to reach goals defined by a business strategy. BPM comprises an emerging IT-supported determination, enhancement, innovation and preservation of end-to-end processes"* [39].

In practice a company introduces business process management to document implicit business processes, automate certain tasks, and optimize internal workflows. Difficulties for a BPM introduction are the chosen degree of granularity, integration into existing workflows and a lack of BPM, and IT knowledge through a company [29].

The continuous process which accompanies with BPM is called BPM lifecycle.

2.1.1 BPM Lifecycle

The BPM lifecyle consists of the steps *design*, *modeling*, *execution*, *monitoring* and *optimization* of business processes [6]. The BPM lifecycle is a never ending looping process. The time to complete each lifecycle step is not completely delimitable. In the following each step is described shortly.

Design: In this step tasks of a business process are identified on a high-level basis, organizational changes are discussed, and additional process dimensions (process participants, notifications, escalations) are defined.

Modeling: The modeling step creates a full documentation of a business process (i.e., by an explicit process model). It is also a validation step, where a specification is formalized (e.g., by utilizing BPMN). In this context even more details, like data handling have to be considered. Formalized process models are validated by simulating certain scenarios (cf. Section 2.2).

Execution: The process model is deployed in a PAIS (cf. Section 2.4). Afterwards, technical details are added in order to achieve the ability to execute the process model. Implementations can be performed with a process notation like BPMN 2.0 and programming languages to develop automatically executed tasks (i.e., script and service tasks) within a process model (cf. Section 2.3).

Monitoring: At this time the implemented and executed process model is monitored against defined business goals. Therefore, often so-called *key performance indicators* (KPIs) are defined to gain convincing facts about efficiency.

Optimization Every preceding step, especially through experiences made by executions, creates suggestions or optimizations, that can be applied to a process model in order to increase its performance. Additional new aspects can show up, which are not covered before. As new requirements are carved out, the BPM lifecycle re-enters the design phase.

2.2 Business Process Modeling

A *business process* describes the rationale of how an organization creates, delivers, and captures value [62]. A *process model* documents a business process and describes which *tasks* have to be done to create a product or service. To build a process model thus it is necessary to be aware of what should be achieved and which participants are involved.

Freund et al. constitute four questions to create a process model [29]:

- Which information is necessary?
- Which detail of information should be chosen?
- How can required information be acquired?
- Which modeling methodology should be used?

To decide which information is necessary a business process can be seen at different point of perspectives. From an **organizational** perspective collaborations between *process participants* of a business process are considered, while a **functional** perspective focuses on which tasks have to be done and in what way these tasks can be organized (e.g. dividing

tasks into subtasks). The **behavioral** perspective defines the order in which these tasks have to be executed to achieve the goal of a business process. Finally, the **informational** perspective describes resources and their production and consumption during the execution of tasks, as well as the informational flow between these tasks. Especially, the question which degree of detail should be chosen, is difficult to answer. Plenty of research exist, trying to give assistance to solve this question [13, 20, 29].

Process models are one possibility to simplify the process of formalizing certain structured tasks. Today they even take care of technical aspects and, if a particular maturity level of detail is reached, process models simply can be transformed into executable business process instances (cf. Section 2.3).

2.2.1 Process Model

A *process model* is represented by a set of *process nodes* and the *control flow* between them. It is a single directed graph, which describes a business process. Process nodes, for example, represent *tasks*, which have to be done, to comply with the process model. The control flow itself concerns about in which temporal chronology these tasks have to be executed.

Further, a process model may include additional *process elements*, like *process data elements*. Every process element has various *attributes* describing its properties (e.g., its name).

2.2.2 Process Modeling Notation

To describe a process model a *notation* is necessary. While the term notation is broadly defined, even a textual document explaining a certain procedure can be seen as process model notation. Therefore, restrictions and guidelines are necessary to define, what exactly has to be the extend of a process model. To achieve the ability for executing a defined process model as process instance, additional information is required.

An example for a simple process model notation are *petri nets* [64]. Particularly, process nodes or tasks refer to transitions and edges—described as control flow—refer to places.

Further, process model notations exist having different scopes, goals, and expressiveness. Examples to be mentioned are *ADEPT2* [68], *Event Process Chains (EPC)* [78] and *Unified Modeling Language (UML) Activity Diagrams* [24]. Generally, the expressiveness of a notation may be determined by applying workflow patterns (cf. Section 2.2.4).

In the context of this thesis, we apply the BPMN 2.0 as business process notation, a de-facto standard for graphical process model notation. A central goal of BPMN 2.0 is to be easily understandable by all process participants (i.e., business analysts, technical developers, and end-users) [30]. Thus, BPMN 2.0 creates a standardized bridge for the gap between business process design and process implementation. Particularly, BPMN 2.0 is an unstructured, graph-oriented process modeling notation. It consists of three different specification sub-classes:

- *Descriptive Conformance Sub-Class (DCS)*: high-level process modeling comprising a subset of all existing BPMN elements
- *Analytic Conformance Sub-Class (ACS)*: detailed modeling, including exception handling and events
- *Common Executable Conformance Sub-Class (CECS)*: all elements

The CECS uses the whole expressiveness of BPMN 2.0. Such BPMN 2.0 defined process models are fully deployable to a PAIS supporting BPMN 2.0 (cf. Section 2.3). CECS consists of the following element types: *Flow Objects*, *Data Objects*, *Connecting Objects*, *Swimlanes*, and *Artifacts*.

Figure 2.1 shows a BPMN 2.0 process model consisting of selected BPMN 2.0 process elements described in the following.

Flow Objects can be further categorized in *tasks*, *gateways*, or *events*. These elements describe the functional nodes of a process model.

Tasks describe actions, a human or system should perform in a business process. Table 2.1 shows different types of tasks and their necessary technical attributes.

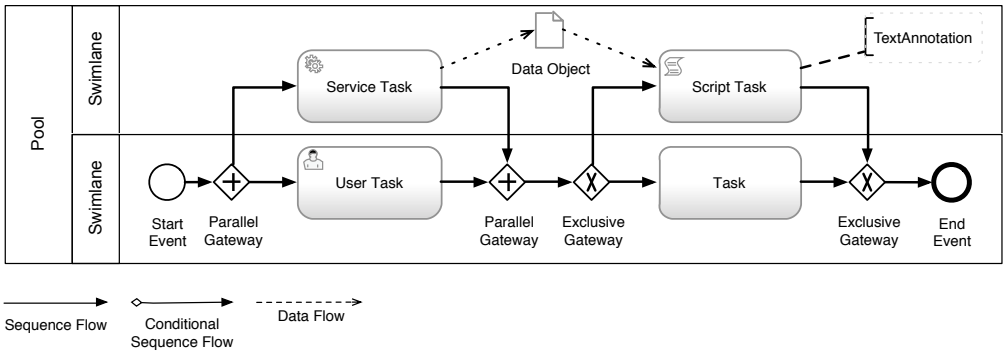

Figure 2.1: BPMN 2.0 Process Model

Element	Description	Attributes
Task	Generic task, which has to be executed by a process participant. This can be either a user or a system component.	id, name
User Task	A *user task* indicates, that a specified user has to interact with the business process. This can be achieved, for example, by providing a user form.	id, name, renderings, implementation, resources, ioSpecification, dataInputAssociations, dataOutputAssociations, loopCharacteristics, boundaryEventRefs
Script Task	*Script tasks* are assigned to specified programming code, which can be automatically executed by a PAIS.	id, name, implementation, operationRef, ioSpecification, dataInputAssociations, dataOutputAssociations, loopCharacteristics, boundaryEventRefs
Service Task	*Service tasks* represent interfaces for interactions with other computer applications and services.	id, name, implementation, operationRef, ioSpecification, dataInputAssociations, dataOutputAssociations, loopCharacteristics, boundaryEventRefs

Table 2.1: BPMN 2.0 Task Types

Gateways describe points, where the control flow is split up by a *split gateway* into two or more branches or the latter is joined together by a *join gateway*. A gateway may express either parallel control flows (i.e., all branches are considered), exclusive control flows (i.e., only one branch is selected for execution), or inclusive control flows (i.e., one or more branches are selected). Exclusive and inclusive gateways need defined decisions to determine the onward progress. Inclusive gateways are not used in this thesis, thus, they are not further discussed (cf. Table 2.2).

Events in BPMN 2.0 consist of *message*, *timer*, and *error events* (cf. Table 2.3). *Message events* indicate, that an external message is sent or received somewhere. Another business process or an intermediate message catch event is able to capture these messages taking

Element	Description	Attributes
Parallel Gateway	A *parallel gateway* indicates a control flow split up. Hence all branches are executed simultaneously.	id, name, gatewayDirection (only converging and diverging)
Exclusive Gateway	In contrast to parallel gateways an exclusive split gateway activates only one outgoing branch. Each branch has a condition attribute splitting each activation into a disjunct activation set. Conditions can refer process data that is available during execution.	id, name, gatewayDirection (only converging and diverging), default

Table 2.2: BPMN 2.0 Gateway Types

Element	Description	Attributes
Start Event	A *start event* triggers the start of a new process instance.	id, name (also conditionalStartEvent, which has an additional conditionalEventDefinition)
End Event	An *end event* is the last element of a process model. It indicates the end of a process path.	id, name

Table 2.3: BPMN 2.0 Event Types

further actions. An intermediate message catch event stops further execution of a process instance, until an expected message has arrived. *Timer events* indicate a delay in the process execution. This can be useful, when further steps of a business process, like sending a dunning letter in domain of accounting, first need a particular interruption.

BPMN 2.0 allows for data flows between tasks. Therefore, Data Objects may be used. A data flow is a process edge indicating a data flow between a Data Object and a certain task. [84] explains, that data flows are not frequently used during process modeling, thus, they are not further discussed in this thesis.

Connecting Objects link above described flow objects to indicate a certain relationship or interaction between them. *Control flow objects* describe the temporal aspects between flow objects, while *message flow objects* and *data association objects* outline a data flow. Associations are used to link artifacts, like text annotations, to other flow objects in order to highlight their affinity.

Pools and Swimlanes are used to define process participants. Pools often represent organizations. Swimlanes again subdivide pools to organize them in smaller units. To give

Element	Description	Attributes
Sequence Flow	A *sequence flow* connects two nodes to indicate a control flow.	id, name, sourceRef, targetRef, default
Conditional Sequence Flow	A *conditional sequence flow* is only activated, when additional condition expression returns true.	id, name, sourceRef, targetRef, conditionExpression ConditionExpression, allowed only for sequence flow out of gateways, may be null

Table 2.4: BPMN 2.0 Connecting Objects

an example, a pool could represent an *Order Service*. This pool can consists of two *lanes*, *Callcenter* and *Delivery Department*.

Artifacts (i.e., Associations, Groups, TextAnnotations) show additional information to the user, which are not directly related to a sequence or message flow. They offer a possibility to document process models more precisely. Artifacts are not further discussed in this thesis, as Section 6 introduces concepts implementing them differently.

2.2.3 Block-Structured Process Models

Block structures known from programming languages are used to capsulate code fragments [51]. In particular, block structures must not overlap, but they can be nested arbitrary deeply. Figure 2.2a shows an example of a process model. The latter is not block-structured, as $SESE1$ does overlap with $SESE2$. In turn, Figure 2.2b is an example of a block-structured process model.

Block $SESE1$ and block $SESE2$ in Figure 2.2b are so-called *Single-Entry, Single-Exit (SESE)* blocks. A SESE defines a subgraph of a process model, where every process node tuple A, B conforms three characteristics: A dominates B, B post-dominates A, and every cycle containing A also contains B and vice versa [37]. In other words, process node A dominates process node B in a process model, if every path from start to A includes B. Further, process node A post-dominate process node B, if every path from B to end includes A. How to determine SESE blocks efficiently is described in [37].

A minimal SESE of a set of process nodes N is a tuple of two process nodes (n_1, n_2), where n_1 denotes the entry of the SESE, n_2 the exit and describes the minimal SESE to surround all process nodes in N. If a process model has to be block-structured, all constraints have to

be regarded (cf. Section 2.2.3). Figure 2.3 shows a process model with its different, nested SESE blocks and a minimal SESE of three nodes (B, C, D).

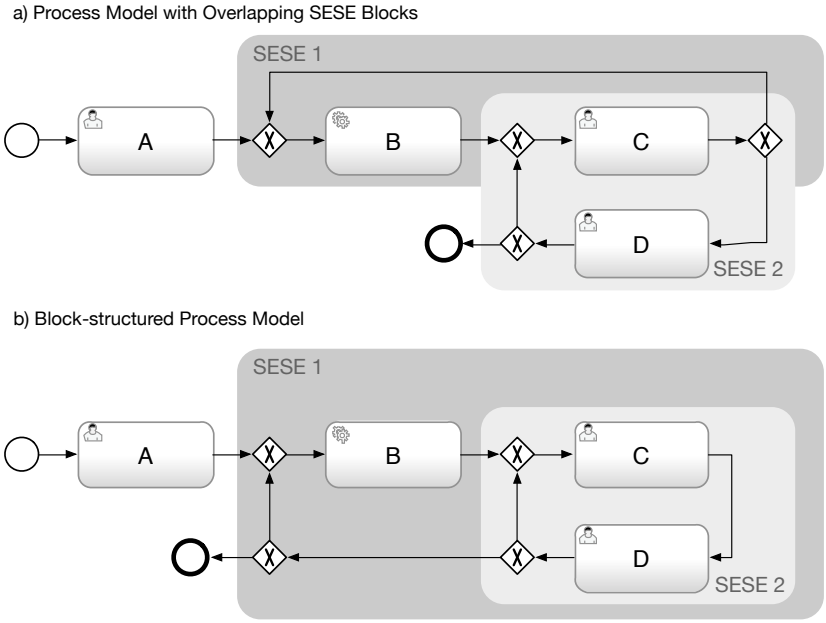

Figure 2.2: Comparism of an Unstructured and Block-Structured Process Model

Block-structured process models are applied in the context of this thesis, because they are the basis for all presented process view operations.

2.2.4 Workflow Patterns and Change Patterns

Process model notations have different cardinality concerning their expressiveness. For example, the BPEL process notation is not able to express an *arbitrary cycle*, while BPMN 2.0 supports them. Arbitrary cycles are cycles, that have more than one entry and exit point. Thus, BPEL is not able to express arbitrary cycles, because it is a block-structured process notation. In [79] so-called *workflow patterns* are introduced, which describe a collection of patterns describing common aspects of process modeling. The latter is divided into different categories (e.g., *control flow patterns* or *advanced branching and synchronization patterns*).

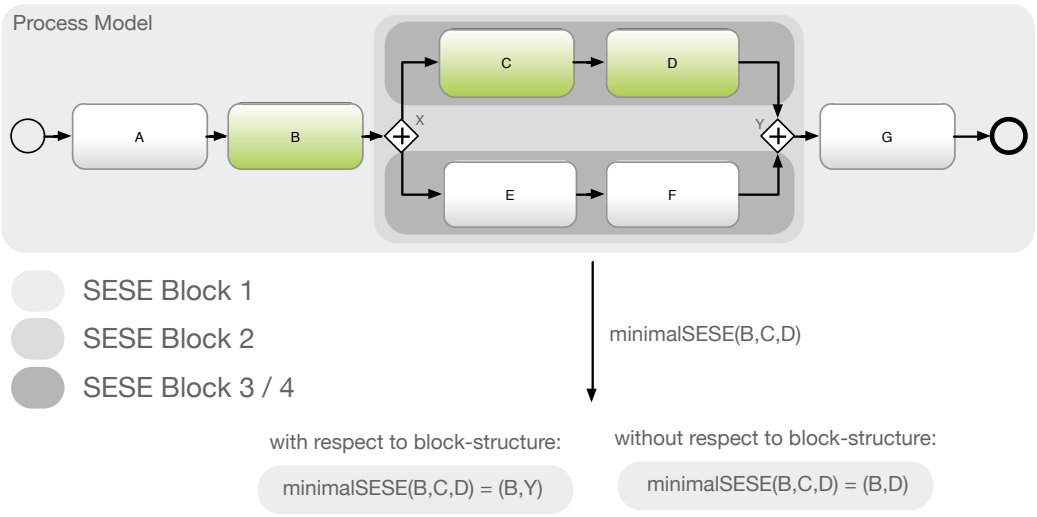

Figure 2.3: SESE Blocks of a Process Model

In order to classify the cardinality of PAIS concerning their ability to change process models, *change patterns* are introduced [81]. Change patterns describe and classify common control flow changes, like the insertion of a new task into a process model.

2.3 Executing Process Models

PAIS offer the ability to execute process models (cf. Section 2.4). Therefore, certain executable code (i.e., the logic of a task) can be attached to tasks, that, in turn, can be executed by a PAIS.

Particularly, BPMN 2.0 offers the ability to model and execute process models. Therefore, several PAIS, like *Activiti* [67] or *jBPM* [17], allow for the direct execution of BPMN 2.0 defined process models.

Thereby, a *process instance* represents a concrete case in the operational business of a company, for example, invoicing a service. From a technical point of view, it is a executable copy of a process model. It contains additional run-time information like an execution state

map of all present executable tasks and a log of already executed tasks. PAIS represent process instances highly memory optimized to handle hundreds or even thousands of process instances. Every business process instance has a business process instance lifecycle with different states [70].

2.4 Process-Aware Information Systems

A *Process-Aware Information System* (PAIS) is a software application platform mapping business processes on a software basis offering the ability to support the BPM lifecycle. As illustrated in Section 2.2, there are different perspectives on business process management. A PAIS takes account of several perspectives, like operation, time, organization, behavior, information, and function, as well as offers different services to support them [70]. It also concerns about a separation of process logic and application code to be run by tasks in a process model.

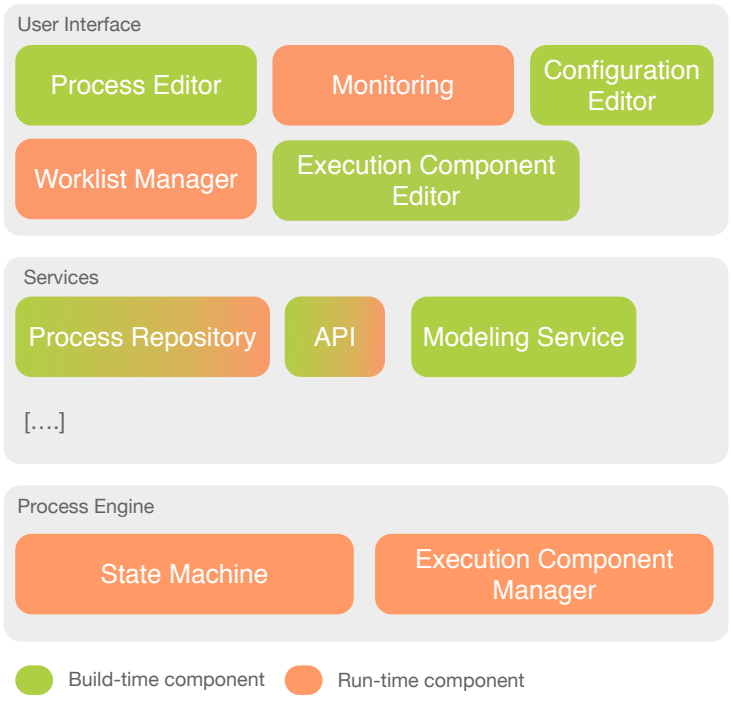

Figure 2.4: Components of a PAIS

A PAIS consists of several components [48]. These are separated logically into *build-time* and *run-time services* (cf. Figure 2.4). At build-time of a process model, a PAIS provides a *process editor*, with which a user is able to define and configure a process model by using a graphical user interface. Resulting process models can then be stored in a *process repository*, as well as its application service components. An *execution component* is a type of software to either execute certain programming code or provide interfaces to other software applications and services to map business logic. Stored process models can be deployed to a *process engine* for creation and execution of process instances. At run-time the *process engine* takes care of all business process instances with different services. It includes, for example, services for service invocation, time management, access control, escalation, logging, and persistence.

2.5 Summary

Business process management describes how companies can reach their business goals by capturing business processes, executing, and managing them. Therefore, processes are described using process models with specific notations. Every notation sets different priorities concerning its representation, cardinality, and purpose. Some process notations, like BPMN 2.0, enable automatic executions on process-aware information systems (PAIS). A PAIS consists of different components to manage and optimize process models. A concept to describe necessary management operations is the BPM lifecycle.

3

Overview on Abstracting Process Models by Applying Process Views

Process models usually comprise tasks from different departments in a company. Furthermore, such process models involve different participants. Particularly, process models often comprise hundreds of tasks, and are hard to understand by non-technical users. Next, changing or evolving process models is challenging and error-prone.

One possibility to reduce complexity of a process model are different process model visualizations, which may change the appearance of process elements. For example, in [42] an approach to transform a process model into a verbalized process description is presented. Hence, a process model in BPMN 2.0 may be transformed into written text comprising a rich english grammar and vocabulary description of all process elements. Furthermore,

there exist other approaches to transform a process models visualization in order to reduce complexity [49, 45].

Another possibility to reduce complexity of process models is *process abstraction*, i.e., only information needed for a specific use case is provided [12, 43]. This can be achieved by applying algorithms to hide parts of the control flow, which are not relevant for the respective process participant. Such process models are also called *process views* [12, 43]. Figure 3.1 shows a process model, which involves five groups of participants: *assistance of photographers*, *photographers*, *copywriters*, *graphic designers* and *layouter*. In particular, a photographer may not interested in tasks of the other participants. Therefore, a personalized process view can be created only showing tasks of the photographer, i.e., tasks, which are located on his pool [9, 10, 11, 69]. However, process views can be arbitrary adjusted to fit a participants needs: an assistant of a photographer should have knowledge of his own and the photographers tasks: its process view would consist of his and the photographers pool. To be more precise, every process element, rather than just complete pools, can be depicted by operations creating a process view. Showing only relevant tasks can also ensure privacy. Process views can be adjusted to participants in a way, only showing such process information they are allowed to see.

Process elements in a process view can also be grouped by so-called *virtual nodes*. The latter represents multiple tasks and may be used to arrange tasks in order to increase clearness of process models. In addition, not only tasks as part of a process model control flow can be hidden, but other process elements, like attributes of tasks, gateways or process data elements. Process view algorithms may also be applied on process instances.

Additionally, it is required to perform changes on process views [41]. Particularly, users must be enabled to modify their own process views. Subsequently, associated process models should be automatically updated in order to keep process view and process model consistent.

In the following, we introduce an approach to enable parametrization of process views allowing for user-specific adjustments and automatic creation.

Therefore, Section 3.1 introduces fundamentals and basic notions. Section 3.1.5 explains how a process view can be generated out of an arbitrary process model. Section 3.1.3 describes advanced techniques enabling updates on process views and propagating them to

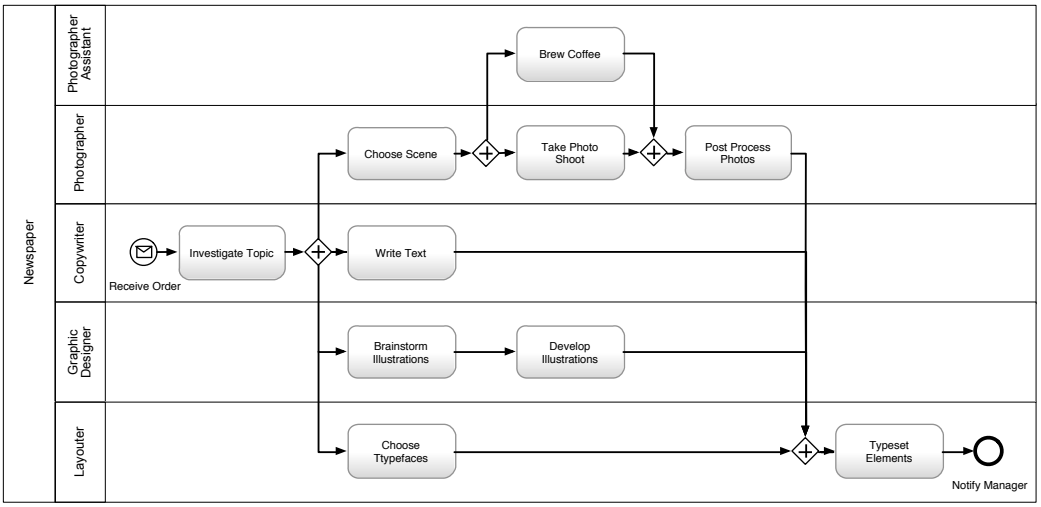

Figure 3.1: Process Model for Article Creation

the linked process model, on which a process view is based on. The last section illustrates the automatic application of process view generation and modification algorithms.

3.1 Fundamentals on Process Views

In the following basic notions, as well as elementary operations to create process views - reduction and aggregation - are outlined, which reduce and aggregate process elements.

3.1.1 Process View

A *process view* is an abstracted process model or process instance, in which not necessary process elements, like control flow elements, are reduced. Further approaches suggest modification or reduction of additional attributes and data elements [44]. However, this thesis focuses on control flow abstraction.

The following definitions are based on the proView project, which enables updating process models as well as all attached process views by applying view update operations on a process view [41]. One essential pre-condition to apply view update operations are block-structured process models or process instances (cf. Section 2.2.3).

Process views are created using a *creation set (CS)*. The latter specifies the schema and appearance of a process view. It consists of a *central process model (CPM)*, a view-specific *change set*, and a *parameter set*.

A *CPM* is a process model on which process views are created on.

A *change set* is a set of *view creation operations*, that are applied on a CPM to calculate a process view out of it. Every *view creation operation* is described by a view creation algorithm and a *node set*, which contains a set of all process nodes involved in a view creation operation. Every view creation operation in a change set has to affect different process nodes than other view creation operations (cf. Section 3.1.4). As a consequence, all view creation operations can be applied in arbitrary order without resulting in differing process views.

The *parameter set* is a set of parameters to control automatic propagation and resolution of ambiguities during process view updates. An automatic propagation is executed after a process view is changed and applied changes affect the associated CPM as well (cf. Section 3.1.3). Ambiguities can occur, when inserting a process node in a process view, where adjacent process nodes are reduced (cf. Figure 3.4). A newly inserted process node D can be set left or right of the reduced process node B. In the example, the parameter `InsertSerialMode` resolves this ambiguity by defining whether the process node is inserted `EARLY` (i.e., before B), `LATE` (i.e., after B), or parallel to process node B.

A parameter set can be defined *globally* for a set of users, *locally* for a specific CPM, or *individually* for every view update operation. By doing so, individual parameter sets have the highest priority and override local and global parameter sets.

If a process model is changed, all attached process views have to be updated as well. Therefore, additional parameters for reduction and aggregation are defined to support automatic updates on attached process views. This procedure is described in Section 3.1.3.

Operation	Description	Operation Type
$RedTask(V, n)$	Reduces a specific process node n in a view V	atomic
$RedSESE(V, N)$	Reduces a set of process nodes N in a view V	compound
$AggrSESE(V, N, v)$	Aggregates a coherent set of tasks N to a virtual node v in view V. All tasks in N have to be part of the same SESE block	atomic

Table 3.1: Control Flow View Creation Operations

An example for a creation set of process view $V1$ in Figure 3.2 is: $CS = \{CPM, ChangeSet = \{RedTask\{B\}\}, ParameterSet = \{\}\}$. Thereby, CPM describes the CPM, where all view creation operations in set $ChangeSet$ have to be applied on. Set $ChangeSet$ contains view creation operation $RedTask$ (cf. Section 3.1.5), which hides task B in process view $V1$, set $ParameterSet$ is empty, because reduction operations do not require parameters.

3.1.2 Control Flow View Creation Operations

Creating process views one or more control flow view creation operations are applied on a copy of the CPM. Such operations act locally, i.e., they do not modify the associated CPM or affect other process views. Table 3.1 shows an overview of control flow view creation operations.

Control flow view creation operations have different characteristics concerning their modification of control flow dependencies between process nodes, like tasks. Control flow dependencies describe temporal and conditional behaviors between process nodes (e.g., the execution order and required conditions to execute a task). Every task has control flow dependencies to any other process node defined in a process model. The simplest case describes, whether a task A has to be executed before, after or parallel to task B.

In [43] a *dependency set* D is defined, which describes control flow dependencies, so-called *dependency relations*, between any two tasks in a process model. A dependency set of the process model in Figure 3.2 before applying view creation operation $RedTask$ is $D = \{(A, B), (A, C)\}$. After applying view creation operation $RedTask(B)$ reducing task B, the dependency set of process view $V1$ shows as follows: $D' = \{(A, C)\}$. Dependency relation (A, B) was removed from D' by view creation operation $RedTask$, this behavior is called *dependency-erasing*. In turn, *dependency-generating* view creation operations insert new dependency relations, *dependency-preserving* do not change the dependency set.

View creation operations *reduce* process elements in the resulting process view. Such reductions can be seen as deleting a process element in a copy of the CPM. Figure 3.2 shows the reduction of task B. Reductions are *dependency-erasing* operations. Operation $RedTask$ is an *atomic* operation and reduces a single task. Atomic operations only affect one process element at once and cannot be further split up. In contrast, *compound operations* combine two or more atomic view creation operations. Compound operation $RedSESE$ reduces a complete SESE block and is a combination of several $RedTask$ operations (cf. Table 3.1). $RedSESE$ is *dependency-erasing* as well.

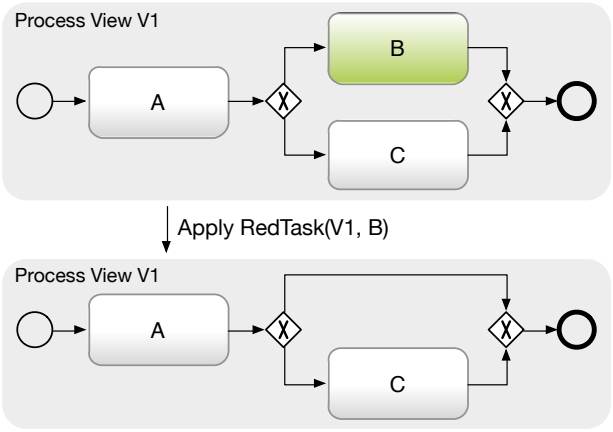

Figure 3.2: Reduction of a Task

Aggregation view creation operations combine a set of process nodes to a single node. The latter is called *virtual node* and can be seen as Sub-process including all nodes aggregated. If process nodes to be aggregated are not in the same SESE block, a least common SESE has to be determined. Otherwise an aggregation would break up the required block-structuredness of the process view. Figure 3.3 shows an example, in which process nodes B and D are aggregated to a virtual node V in the resulting process view. Both nodes are not part of the same SESE block. To be able to aggregate them, a least common SESE block has to be determined by applying operation minimalSESE(B,D). The latter results in process nodes $\{X, Y\}$. Aggregation operations are *dependency-generating*.

Atomic view creation operations comprise a node set containing every process node by the view creation operation. Such a node set is called *dedicated node set*. In contrast, node sets of compound view creation operations, like operation $RedSESE$, only contain the entry

and exit node of a SESE block, which is called *abstract node set*. Applying such operations on the CPM requires to calculate all process nodes described by the abstract node set.

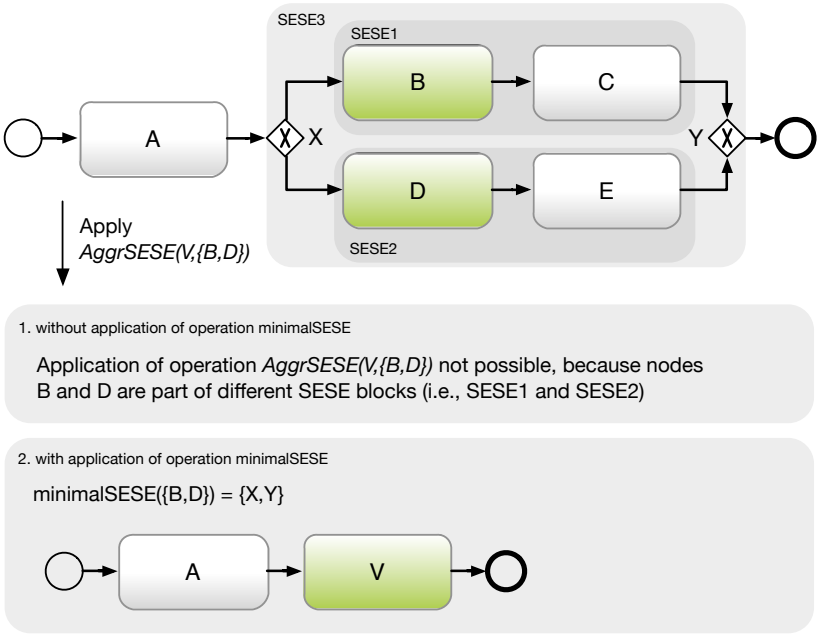

Figure 3.3: Aggregation of Tasks

Applying view creation operations may result in unnecessary process fragments, like empty branches or a completely empty branching block. Such process fragments can be removed by executing refactoring operations (cf. Section 3.1.5).

3.1.3 Control Flow View Update Operations

Process models have to be changed often as a result of amended business goals or business situations (e.g., by optimizing internal working steps). Changing large process models with dozens of tasks is complex, and thus error-prone. Process views personalized for a process participant reduce complexity. In order to allow changes on process views directly, update operations for process views, so-called *view update operations*, are described subsequently [41].

Control flow changes (e.g., inserting a task) can be applied on a CPM or on an associated process view. Every associated process view has to be adapted after changing a CPM. Thereby, it has to be decided, whether control flow changes, except deletions, on a process model are shown or not adducing the parameters `AggrComplMode`, `AggrPartlyMode`, `RedComplMode` and `RedPartlyMode`. Figure 5.2.3 shows an example illustrating all four parameters. If an updated process node is completely surrounded by already aggregated process nodes, parameter `AggrComplMode` defines, if the aggregated SESE remains reduced (i.e., parameter is set to value `AGGR`) or will be revealed (i.e., `AggrComplMode-SHOW`). Parameter `AggrPartlyMode` further proceeding, when only one adjacent flow node is aggregated. Corresponding parameters can be set for reduction operations (i.e., parameters `RedComplMode` and `RedPartlyMode`).

Additionally, updates can be executed on process views directly, which, in contrast to view creation operations, also propagate changes to associated CPMs. This enables participants to alter a process view, while keeping the underlying CPM and all associated process views up-to-date.

One necessity is the ability to propagate process view updates to the underlying CPM as well as to propagate updates automatically. One problem to be solved is, that changes on process views may generate *ambiguities* in relation to the associated CPM, which also have to be resolved automatically.

Table 3.2 describes basic control flow update operations for process views. Column *Parameter* describes respective parameters, which may be defined in a parameter set in order to resolve ambiguities for the specific update operation.

To demonstrate arising ambiguities, Figure 3.4 demonstrates a task insertion. $InsertSerial(V, D, A, C)$ adds a task D between task A and task C in a process view $V1$. Propagating this change to the associated CPM results in a decision problem: task B, which was reduced in process view V, is present in the CPM. Thus, $InsertSerial(V, D, A, C)$ has to be translated in either inserting task D between task A and task B, or between task B and task C. To be able to solve this ambiguity parameter `InsertSerialMode` decides, if inserting a task is executed in immediate vicinity to the most left (`EARLY`) or most right aligned process nodes (`LATE`).
If a complete block shall be inserted by using $InsertParallel$ / $InsertConditional$ / $Insert$-

Loop, two decisions (for every split gateway) have to be solved. Therefore, `InsertBlockMode` offers four values indicating the correct insertion point for the split gateway (`EARLY_`, `LATE_`) and join gateway (`_EARLY`, `_LATE`).

After all, associated views $V2$ and $V3$ are updated according to their parameters. Automatic process model *change propagation* to other process views is determined by different parameters to decide, whether the incoming process model change is shown in other process views or aggregated / reduced accordingly. These parameters also differentiate between partial (i.e., parameters `AggrPartlyMode`, `RedPartlyMode`) and complete intersection (i.e., parameters `AggrComplMode`, `RedComplMode`). The latter can be set to either show or hide propagated process model changes in other process views associated with the changed CPM.

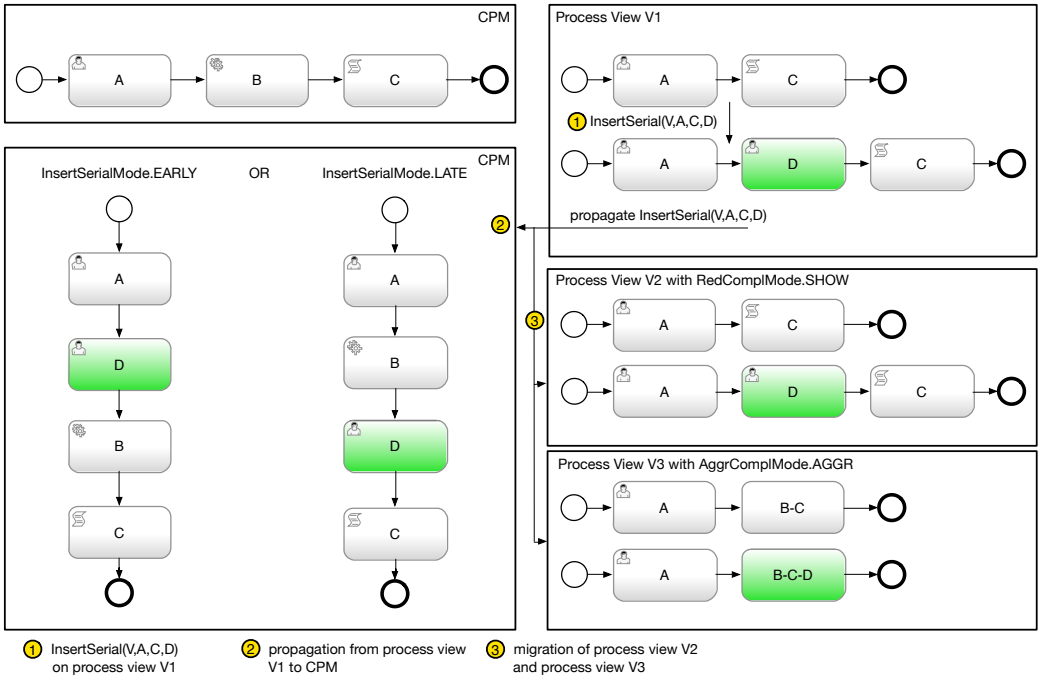

Figure 3.4: InsertSerial Update Operation

Operation	Parameter	Description
$InsertSerial(V, n, n_p, n_s, InsertSerialMode)$	InsertSerialMode = {EARLY, LATE, PARALLEL }	A new node is inserted as early or late as possible—or parallel to reduced nodes
$InsertParallel(V, n, n_p, n_s, InsertBlockMode)$ $InsertConditional(V, n, n_p, n_s, InsertBlockMode)$ $InsertLoop(V, n, n_p, n_s, InsertBlockMode)$	InsertBlockMode = {EARLY_EARLY, EARLY_LATE, LATE_EARLY, LATE_EARLY }	Similar to InsertSerialNode, but consists of four parameters to adjust an insertion of every gateway (split and join) detached
$InsertBranch(V, e, n_p, n_s)$		Inserts a new branch
$DeleteTask(V, n)$	DeleteTaskMode = {LOCAL, GLOBAL}	Deletes a task only in a view (semantically equivalent to RedTask) or in its corresponding CPM
$DeleteBranch(V, e)$		Deletes a branch
$DeleteBlock(V, n_{start}, n_{end}, DeleteBlockMode)$	DeleteBlockMode= {INLINE, DELETE}	Deletes a block by deleting its surrounding gateway nodes and serializing all branches in series (INLINE) or deleting the block with all nested nodes

Table 3.2: Control Flow View Update Operations

3.1.4 Migration of Process View Change Sets

All view creation operations in a change set have to be disjunct, i.e., a process node has to be present in no or exactly one node set. One advantage is, that view creation operations can be applied on a process model in arbitrary order.

Change set *migrations* become necessary, if any two view creation operations show intersecting node sets, or if control flow changes on a CPM are propagated to associated views. A need for migration can be determined by processing all node sets in a change set.

Intersecting node sets can be divided into four scenarios:

- Reduction of a virtual node (i.e., `RedOnAggr` scenario)
- Further aggregation of one or more virtual nodes (i.e., `AggrOnAggr` scenario)
- Aggregation of process nodes surrounding one or more reduced process nodes (i.e., `AggrOnRed` scenario)

- Reduction of process nodes surrounding one or more reduced process nodes (i.e., `RedOnRed` scenario)

Solving cut sets of reduced or aggregated process nodes depends on the available view creation operations (cf. Section 3.1.2) and node set type (i.e. dedicated or atomic node sets, cf. Section 3.1.1), because node sets either contain all affected nodes (dedicated node set, used by atomic view create operations) or just entry and exit nodes of an affected SESE block (abstract node set).

`RedOnAggr` can always be solved by removing the interfering aggregation operation and migrating all flow nodes contained in the aggregation node set to separate reduction operations. All other scenarios are node set type dependent.

When using *dedicated node sets with atomic view creation operations*, all overlapping view creation operations can be easily migrated. Scenario `AggrOnAggr` can be solved by removing all interfering aggregation operations and include their node set to the new aggregation operation. When scenario `AggrOnRed` occurs, all involved reduction operation have to be removed and their node set has to be migrated to the new aggregation operation. Scenario `RedOnRed` is not possible when using dedicated node sets, because atomic creation operations do not cover such case.

Migration on *abstract node sets with compound view creation operations* works similar to dedicated node sets, but every intersection has to be computed by analyzing a node set SESE block and either adjustment of a SESE blocks border nodes (i.e., *partial intersection*) or deletion of a complete view creation operation (i.e., *complete intersection*). *Partial intersections* occur, if an affected SESE block of one view creation operation is not fully covered by the other view creation operation's affected SESE block. If a SESE block completely surrounds another SESE block, the latter is called *complete intersection*.

If a change set migration results in an empty node set, the whole view creation operation is deleted, as its application would show no effect.

3.1.5 Process View Refactoring Operations

Applying a creation set on the CPM may create redundant or not necessary control flow elements, which may be removed by *refactoring operations*. Refactoring operations are semantic-preserving, thus, an application on a process model (e.g., a CPM or a process view) does only remove control flow elements beside the point [80]. For example, an application of refactoring operations on a process model in Figure 3.2 has no effect on its process elements, because conditional gateways in a control flow have to be preserved in order to ensure behavioral equality. In the example, task C is executed conditionally - by removing the surrounding gateway block we would create the impression, that task C has to be executed on any account.

We focus on three refactoring operations: $SimplifyEmptyBranches$, $SimplifyEmptyBlocks$, and $SimplifyMultipleBlocks$ (cf. Table 3.3).

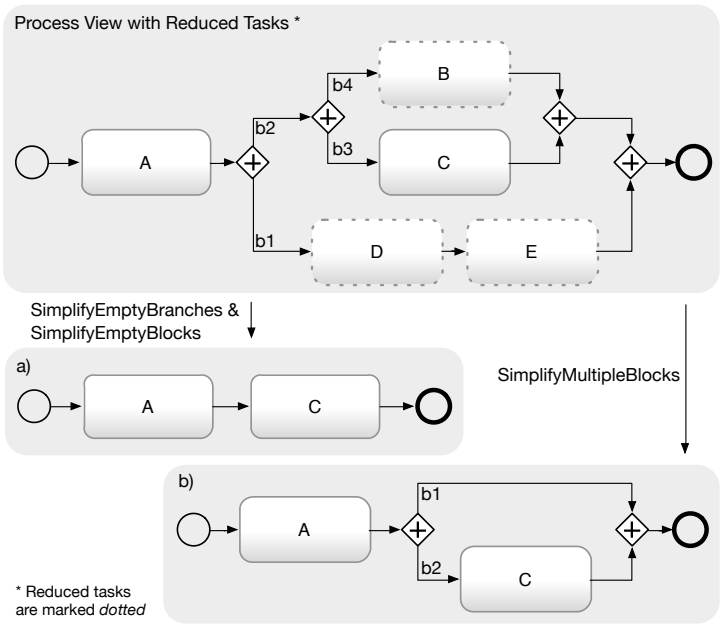

Figure 3.5: Refactoring Operations

Refactoring operation $SimplifyEmptyBranches$ removes empty branches of parallel branching gateways. Refactoring operation $SimplifyEmptyBlocks$ removes empty blocks from

Operation	Description
$SimplifyEmptyBranches(P)$	Removes empty branches of parallel branching gateways in a process model P.
$SimplifyEmptyBlocks(P)$	Removes SESE blocks only consisting of a split and a join gateway in a process model P.
$SimplifyMultipleBlocks(P)$	Simplifies multiple SESE blocks by condensing nested, directly adjacent gateway nodes of the same type to a SESE block with only two appropriate gateways (i.e., split and join gateway) in process model P.

Table 3.3: Process View Refactoring Operations

a process model. An empty block is a process fragment of a process model consisting of two gateway nodes and their corresponding control flows. Refactoring operation $SimplifyMultipleBlocks$ reduces nested, directly adjacent gateway nodes of the same type to a fragment with only one gateway block.

Figure 3.5 illustrates the application of the three refactoring operations on a process view with reduced tasks: by applying $SimplifyEmptyBranches$ in combination with $SimplifyEmpty-Blocks$, first, empty branches $b1$ and $b4$, then the outer gateways are removed (cf. Figure 3.5a). The application of $SimplifyMultipleBlocks$ leads to a removal of the inner gateway block, consisting of two parallel gateways, branch $b4$ and task C, is removed, while task C is preserved (cf. Figure 3.5b).

Attention should be paid to branch conditions used with conditional gateways [10]. The latter splits up a process models control flow by *conditions* assigned to each branch. At run-time, the branch to be activated has to be decidable. Therefore, all condition dimensions (i.e., dimension of a variable in a condition) are usually covered. An application of $SimplifyEmptyBranches$ may remove empty, conditional branches resulting in a loss of a complete dimension coverage. Hence, all remaining branch conditions have to be adjusted to the effect, that the latter cover all condition dimensions.

A software architecture for creating process views is proposed in [40]. It consists of a *visualization engine* and a *change engine*. The *visualization engine* is responsible for creating process views, while changes on process views are processed by the *change engine*. Figure 3.6 is divided into a view client, a view server and an underlying PAIS. The client is responsible to render and show process views, while the server manages creation sets and dispatches incoming updates from process view clients. If a view update operation is applied on a process view, first the CPM is updated. Therefore, a process modeling component applies process model changes on the CPM. Second, refactoring operations are

applied on the CPM (cf. Section 3.1.5). Then, all change sets are migrated by a change set component (cf. Section 3.1.4). Finally, changes on the CPM are propagated to all associated views, either by only processing changes, or recreating all views. Different parameters can be set to either process, or discard incoming process model changes (i.e., parameter `{Aggr,Red}PartlyMode`, cf. Section 3.1.3). In order to create a process view, the CPM to be associated is duplicated. Next, the creation set is applied on the process model. Finally, refactoring operations are executed on the process view (cf. Section 3.1.5).

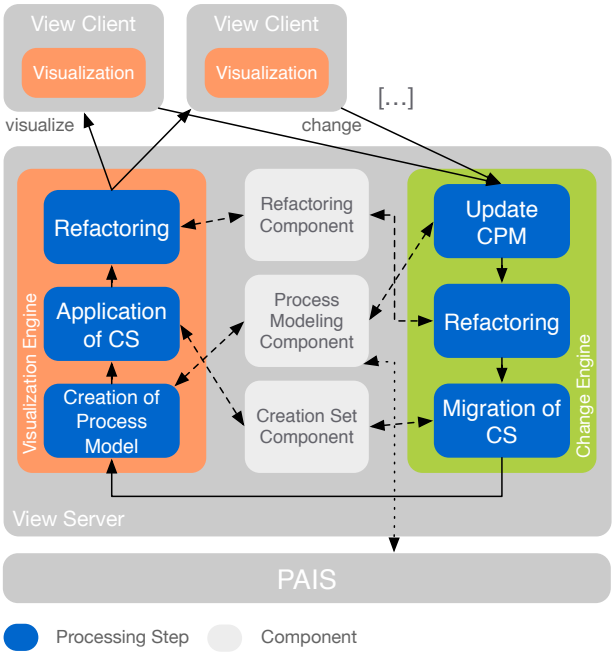

Figure 3.6: Application Schema for Process Views

3.2 Discussion

Process views, as presented in this section, allow to hide and condense process nodes based on the creation operations *reduction* and *aggregation*. Additionally, process views may be updated by respective view update operations (e.g., view creation operation $RedTask$). View update operations, in combination with parameter sets and process view migrations,

allow process participants to create and update their own process view, while the associated CPM and other process views are held up-to-date.

The approach presented in this section lacks of a CPM-independent description of process views to decouple a creation set from a specific CPM. Instead, a creation set is valid for one specific process model. Additionally, process views are created using a creation set, whose definition follows by dedicating view creation operations. As a consequence, it is not possible to define process views based on an abstract definition (e.g., *show only tasks a particular process participant is involved*). Additionally, change set migrations are costly, because every time a view operation is executed, their necessity has to be determined by considering all node sets in a change set. Summarizing, there is a need for process view definitions, that may be applicable on arbitrary process models and allow for defining view creation operations based on aspects (i.e., *show only tasks matching particular conditions*).

Furthermore, there is a lack of a comprehensive software implementation for process views supporting CPM-independent definitions. In order to address these issues, the next section shows up requirements for such a CPM-independent language.

3.3 Requirements for a BPM-Specific Language

As discussed in Section 3.2, no approach exists to define process views independently from a specific process model. In addition, process model changes are dependent on a process modeling notation and a PAIS. Hence, they cannot be easily exchanged between them.

Other domains, like *relational database management systems (RDBMS)*, provide dedicated languages for data definition and data manipulation (e.g., *Structured Query Language (SQL)* [36]). In particular, SQL is a system independent language for creation, modification, and deletion of data stored in a RDBMS. SQL is able to define new data structures imperatively or refer to existing data by describing changes relational to existing data. In particular, SQL is independent from technical implementations.

SQL is a so-called *domain-specific language (DSL)*. In contrast to a *general-purpose language*, like the programming language Java, a DSL has limited expressiveness to support one aspect of a domain. If proper implemented, a DSL often offers a higher run-time

efficiency, since it can be implemented more efficiently utilizing specialized optimizations than a general-purpose language [76].

There are two types of DSLs: an *internal* DSL is a subset of a general-purpose language and uses its expression structures. One example is Rails[1] for Ruby. In contrast, an *external* DSL is a separate language from the main language of the application it is based on. Thus, it offers a custom syntax. SQL is an example for an *external* DSL.

A DSL for business process management may solve discussed issues. In the following, requirements for a BPM-specific DSL are outlined (cf. Table 3.4).

A goal of a DSL for process models is to be able to support the BPM lifecycle (cf. Requirement GRQ-1). A generic process modeling notation is required by a DSL to support typical graph-based process modeling notations (cf. Requirement GRQ-2). The DSL should be able to find and retrieve process models stored in a process repository. Therefore, search expressions offer the ability to find process models (cf. Requirement GRQ-3). A resulting requirement for process data discovery are dependency representations defining, how search expressions may define relations between different process models and process elements (cf. Requirement MRQ-2).

In order to ensure a mapping between any process modeling notation and the DSL's process modeling notation, an underlying meta-model of the DSL has to be able to describe any type of business process (cf. Requirement MRQ-1). A modular concept should enable extensibility of supported process modeling notations, process model change operations, and the DSL's internal processing (cf. Requirement GRQ-5).

Process elements discovered by the DSL should be updatable. Therefore, operations to change and store an updated process model should be provided by the DSL (cf. Requirements MRQ-3 and MRQ-6). Provided operations should only apply changes supported by the initial process modeling notation. Therefore, the DSL has to be able examining their expressiveness (cf. Requirement MRQ-4). Often companies with large process models require complex operations managing these models, which should also be definable and applicable by the DSL (cf. Requirement MRQ-5). The DSL should support process views based on view creation operations and custom process view definitions to simplify managing

[1] open-source web framework based on the Ruby programming language

and displaying those large process models (cf. Requirements VRQ-1 and VRQ-2). Updates on created process views further simplify their handling and, thus, should also be supported (cf. Requirement VRQ-3).

3.4 Summary

This section introduces process views, which abstract process models by only showing necessary process elements in order to simplify the handling of large business processes with dozen of process elements. Hence, process nodes may be aggregated or reduced. Arising problems, when applying these view creation operations, are solved by utilizing refactoring operations and change set migrations. Additionally, process views may be changed by applying view update operations. Process models related to a process view are automatically adjusted to keep these models consistent.

Currently, no approach exists for a process model independent definition of process views. Furthermore, a creation and update of process models is dependent from a process modeling notation and PAIS. In this section requirements for a BPM-specific language were verbalized. These include proposals to enable process data discovery, generic process model definitions, process views, and interoperability with any PAIS.

Requirement	Description
General Requirements	
GRQ-1 Support Complete BPM Lifecycle	The DSL should enable full support for every BPM lifecycle phase.
GRQ-2 Generic Process Modeling Notation	Introduction of a generic process modeling notation, which is able to represent ideally each graph-based process notation.
GRQ-3 Process Data Discovery	Find process models based on search expressions, like process similarities.
GRQ-4 Correctness	The DSL should provide correctness checks and abilities to define conditions process models have to comply with. Additionally, arising ambiguities during process model changes should be solved automatically, the application of change operations corrupting a process model should be prevented.
GRQ-5 Modularity	The DSL's expressiveness has to be extensible, which comprises the internal processing, supported process modeling notations, search and process model change operations.
GRQ-6 Interoperability	The DSL should be defined independently from a specific PAIS.
GRQ-7 Graph Processing Operations	The DSL should provide graph processing operations applicable on its meta-model implementing common graph algorithms (e.g., calculation of predecessors and successors of a process node). Graph processing operations may be used by all DSL-provided operations (cf. Requirement MRQ-3) or supplied extensions (cf. Requirement GRQ-5).
Process Modeling Requirements	
MRQ-1 Process Model Definition	The DSL should be able to declare process landscapes, process models, process fragments, and process elements.
MRQ-2 Dependency Representation	The DSL should allow to express dependencies between different process models, fragments and process elements. This functionality is, for example, a condition for requirement GRQ-5.
MRQ-3 Process Model Change	Changes on process models should be supported by the DSL.
MRQ-4 Process Model Expressiveness	An approach to describe a process notations cardinality has to be provided by the DSL. This requirement is needed to check for compatibility with process model changes (cf. Requirement MRQ-3).
MRQ-5 Custom Process Model Change Definition	The DSL should allow for custom process model change definitions. Therefore, custom process model changes should be able to utilize all DSL-supplied operations.
MRQ-6 Persistence Behavior Manipulation	Allow to modify processing between the PQL framework and any PAIS.
Process View Requirements	
VRQ-1 Process View Creation	Process view creation operations have to be supported by the DSL (cf. Section 3.1.2).
VRQ-2 Custom View Creation Definition	The DSL should allow for custom view creation operations. Therefore, the latter should be able to utilize all DSL-supplied process view creation operations.
VRQ-3 Updates on Process Views	Process view update operations have to be supported by the DSL (cf. Section 3.1.3).

Table 3.4: Requirements for a BPM-specific DSL

4

Process Query and Modification Language

Companies often have dozen of process models, which have to be up-to-date. Various PAISs across a companies IT infrastructure complicate changes on process models, which may be shared across different PAIS. Furthermore, every PAIS supports different sets of process modeling notations and implements methods changing process models in a different manner.

In the following the *Process Query Language (PQL)*, a simple data definition and data manipulation language "for BPM needs" is introduced, based on requirements verbalized in section 3.3. PQL supports the definition of process models and process views, changes on process models and process views and is able to discover and retrieve process models from process repositories. PQL may be used as query language embedded in a request-

response[1] protocol, as configuration language to define process model changes or as generic interchange format for process models.

Section 4.1 describes the PQL meta-model required to support process modeling notation-independent changes on process models. Furthermore, concepts to ensure a process models correctness are presented. Section 4.2 describes concepts of PQL including the PQL meta-model, operations to change process models, concepts enabling discovery of process models in a process repository, or the definition of process views. Section 4.3 introduces a software architecture supporting PQL. Section 4.4 concludes this chapter.

4.1 PQL Process Meta-Model

To comply with requirement GRQ-2 (i.e., generic process model representation), process models may be either described by a generic process model or by a pre-processed data structure, like a process structure tree.

In [75] a generic meta-model is proposed, which divides a process model in the perspectives: functional, behavioral, organizational and informational (cf. Table 4.2). A *process definition* PD based on such a meta-model is defined by a tuple $PD = (Elements, ControlFlow, ProcessLogic)$. Set $Elements$ consists of process elements describing a non-behavioral perspective (i.e., tasks, process participants, or process data elements). Furthermore, set $ControlFlow$ comprises behavioral process elements, i.e., *operators* and *connections*. Operators denote process elements influencing a control flow (i.e., gateways), while connections describe control and data flow edges. Finally, set $ProcessLogic$ defines relations between behavioral and non-behavioral elements.

Converting process models of different process modeling notations into a generic process modeling notation and vice versa might be complicated due to different process elements [38]. For example, exclusive split gateways of a control flow are described by different process elements. Furthermore, exclusive gateway conditions may also have different representations depending on the process modeling notation. Particularly, the latter can be described by different script languages, i.e., UEL or JavaScript. However, automatic

[1] message exchange pattern [35]

Attribute	Description
Generic Attributes	
identifier	Identifies a particular process element from others, has to be unique (e.g., identifier="nid4").
name*	Name of a process element presented to a process participant (e.g., name="Store Invoice").
extensions*	Set of attributes to store additional information, for example, process data values (i.e., if a process node class is INFORMATIONAL) or a human-readable documentation for a task (e.g., extensions={documentation="Stores the created invoice in the database."}).
Node Attributes	
nodeClass	Distinguishes between tasks, control flow modifying nodes and additional entities (like process participants and process data elements). Send and receive events are distinguished based on LINK edge directions. Possible values are: FUNCTIONAL: process tasks BEHAVIORAL: gateway and event representation ORGANIZATIONAL: process participants INFORMATIONAL: process data element representation
nodeType	Further distinguishes a node class into different node types. Attribute nodeType is dependent from nodeClass. Possible values are: FUNCTIONAL: USER (involves a human), SERVICE (executed by a external service), SYSTEM (executed by the PAIS) BEHAVIORAL: AND,OR,XOR,EVENT ORGANIZATIONAL: ENTITY, GROUP INFORMATIONAL: not applicable
Edge Attributes	
edgeClass	An edge class defines, whether edges are behavioral-relevant (control flow) or denote relations between entities (organizational, informational nodes). Possible values are: FLOW: connects FUNCTIONAL and/or BEHAVIORAL nodes LINK: connects ORGANIZATIONAL and/or INFORMATIONAL nodes with FUNCTIONAL and/or BEHAVIORAL nodes
edgeType*	Classifies edges more precisely. Attribute edgeType is implementation-dependent.

* Attribute is optional

Table 4.1: PQL Meta-Model Element Attributes

conversion of semantical meanings is a far complex topic and is not further discussed in this thesis.

The PQL process meta-model is a directed graph consisting of nodes and edges. Each node and each edge has respective attributes, which describe, for example, its functional behavior. Each node has a set of *element attributes*, which can be mapped to a certain process models notation (cf. Table 4.1). Element attributes are mandatory in order to ensure, that changes on process models defined by PQL have sufficient information to be executed unambiguously. Furthermore, the set of element attributes must be extendable to cover the complete cardinality of a process modeling notation. *Custom element attributes* are PAIS-specific and can only be covered by providing operations to define and modify these additional attributes.

Table 4.1 shows PQL element attributes comprising all perspectives presented in Table 4.2. Additional perspectives, like time, error, or operation (e.g., process instance behavior), are not considered.

Every instantiation of a meta-model element is assigned to generic attributes, which are mandatory in order to identify nodes and edges unambiguously. *Node-specific attributes* comprise attributes *node class* and *node type*. Attribute node class offers a high-level classification of process nodes, which is necessary to identify nodes by their functionality (e.g., BPMN 2.0 tasks are represented by `FUNCTIONAL` nodes). Node types are used, for example, to distinguish between different task types (e.g., user and service tasks). Edges are divided into classes and types as well: *edge class* `FLOW` denotes a control flow, edge class `LINK` describes data or message flows between process nodes. Edges are directed and may influence the semantics of a node: a `LINK` edge leaving a `BEHAVIORAL` node with node type `EVENT` denotes the process node as message-throwing event.

Process participants are representable by nodes having attribute value `ORGANIZATIONAL` for attribute node class. To define dependencies between process participants and nodes of type `FUNCTIONAL`, an organizational node can be linked to a functional node by an edge with edge class `LINK`. This method does not differentiate between process participants allowed to execute a task or designated participants. In order to define a fine granular relationship between participants and tasks, attribute *edge type* may be used. This attribute is PAIS-specific, standard operations provided by PQL only take edge classes into account, but can be overwritten or extended accordingly (cf. Section 4.2.4).

Organizational and informational perspectives are represented by nodes to avoid redundancies and ambiguous identification of such elements. These nodes may be also represented by adding additional attributes to functional and behavioral nodes, e.g., an attribute of a process participant designated to execute a specific task. However, this increases, for example, the effort to identify a process participant's designated tasks, because each node defining this attribute has to be considered. Additionally, view creation algorithms can be implemented more efficiently, as affected nodes are identified by following all links instead of exploring a process model's whole set of process nodes. For example, if we want to reduce all tasks in a process view where a specific process participant is not involved in, every task in a process model's node set has to be treated and checked (i.e., if this task contains an attribute describing the process participants involvement). By defining a process participant

Perspective	Description	Meta-Model Elements
Functional	Describes all elements, that execute a certain task	Event, Task, Sub-Process
Behavioral	Defines the order, in which functional elements have to be executed	Control flow elements, divided into operator and connection. An operator defines control-flow conditions, a connection represents a link between two meta-model elements
Organizational	This perspective describes participants involved in a process	Participant
Informational	Resources, that are produced or consumed during an activity execution	Resource

Table 4.2: Perspectives of a Process Model

as separate process element linked to its involved tasks, a participant's tasks can be simply identified by analyzing on which tasks the element is linked to.

Additionally, our meta-model is able to exchange process models between different PAIS. In particular, there exist plenty of process model exchange formats [55], whereas different PAISs support only a subset. When transferring a process model from one PAIS to another, it often has to be re-modeled in the target PAIS due to poor support of such process model exchange formats. By using our meta-model as exchange format, only implementation-specific process data has to be re-modeled. In addition, parts of the specific process data, like branch conditions may be translated automatically, which further minimizes adaption effort. Particularly, cross compilers, for example, are able to convert Java into JavaScript code.

4.1.1 Process Model Correctness and Expressiveness

Process model notations have different cardinality concerning their expressiveness. The expressiveness of a process model may be determined by the amount of supported workflow patterns (cf. Section 2.2.4). In order to distinguish different cardinalities, PQL offers *expressiveness descriptions* for process model notations and *restrictions* for specific process models.

Expressiveness descriptions are used by change or view creation operations of PQL to decide, whether the latter can be applied or not. An expressiveness description contains a list of workflow patterns supported by a specific process model notation and how they may be applied (cf. Section 2.2.4). For example, workflow pattern *exclusive choice* is

supported by BPMN 2.0 and may be mapped by a conditional split gateway. Thus, the expressiveness description for BPMN 2.0 contains a tuple $SupportedPattern(P, M)$ consisting of the workflow pattern $P = CFPatternExclusiveChoice$ and a *mapping description* M (cf. Section 4.1.2). In the example, M is a description to convert a BPMN 2.0 conditional split gateway into the corresponding PQL process element (i.e., a PQL process node with nodeClass=BEHAVIORAL and nodeType=XOR).

Restrictions limit operations applicable on specific process models. One example for a restriction is a block-structured layout to be able to apply process view update operations (cf. Section 3). Both, expressiveness descriptions and restrictions, are essential for change and process view update operations, because not supported changes would potentially invalidate a process model.

4.1.2 PQL Process Model Mapping

In order to support different process model notations (cf. Requirement GRQ-2), PQL has to provide concepts to map a process model represented by a dedicated process model notation into PQLs meta-model and vice versa. Basically, such a mapping can be conducted with a limited set of expressiveness, because process model notations are mainly based on directed graphs. However, if a semantical behavior of process elements has to be described (i.e., exclusive gateways splitting up a control flow based on defined conditions), each element of a notation has to be mapped to the PQL meta-model representation, because each process model notation differs in its representation.

PQLs meta-model, therefore, offers predefined constructs to describe functional, behavioral, organizational, and informational perspectives. Process element properties describing semantical behaviors of different perspectives are a fundamental requirement in order to execute PQL-supplied process model change operations correctly.

Mappings between dedicated process model notations and the PQL meta-model may be described by using a *mapping description (MD)*. It consists of a PQL process fragment and another process model notations fragment describing the same in order to support a conversion between them.

Figure 4.1 illustrates a mapping between a BPMN 2.0 process model and its corresponding PQL process model. The start event in the BPMN 2.0 process model with identifier=1 is mapped as process node with node class attribute value `BEHAVIORAL` and node type attribute value `STARTEVENT`. Tasks are mapped as node class attribute value `BEHAVIORAL` with node type attribute values `userTask` (e.g., node with identifier=2) or `serviceTask` (e.g., nodes with identifier=3 and identifier=5). Typically, technical implementation details in process models are described by String values, which either contain scripts or references to methods to be invoked in order to execute each task. As the PQL meta-model supports notation-dependent node attributes modifiable by PQLs change operations, these details can be easily added and managed. Mappings for other notations can be performed the same way as featured in Figure 4.1.

4.2 Process Query Language

This section describes different concepts considered by PQL. First, an approach to discover process models from process repositories is presented. Subsequently, process model change operations supported by PQL are described. Furthermore, an approach to enable process views on arbitrary process models is described. Finally, the PQL modularity concept is explained, which offers capabilities to extend various functionalities of PQL.

Figure 4.2 shows an overview of different PQL concepts. PQL is able to discover process models in a PAIS process repository. After retrieval a process model represented by a different process modeling notation can be mapped by PQL's meta-model (cf. Section 4.1). A PQL process model, thus, may be changed by change operations described in section 4.2.2 and further abstracted by view creation concepts explained in section 4.2.3. PQL's modularity concept allows for extending all components, which is described in section 4.2.4.

4.2.1 Process Model Discovery Representation

Discovering process models in process repositories requires the definition of search conditions. Search conditions, in turn, require the ability to express *dependencies* between

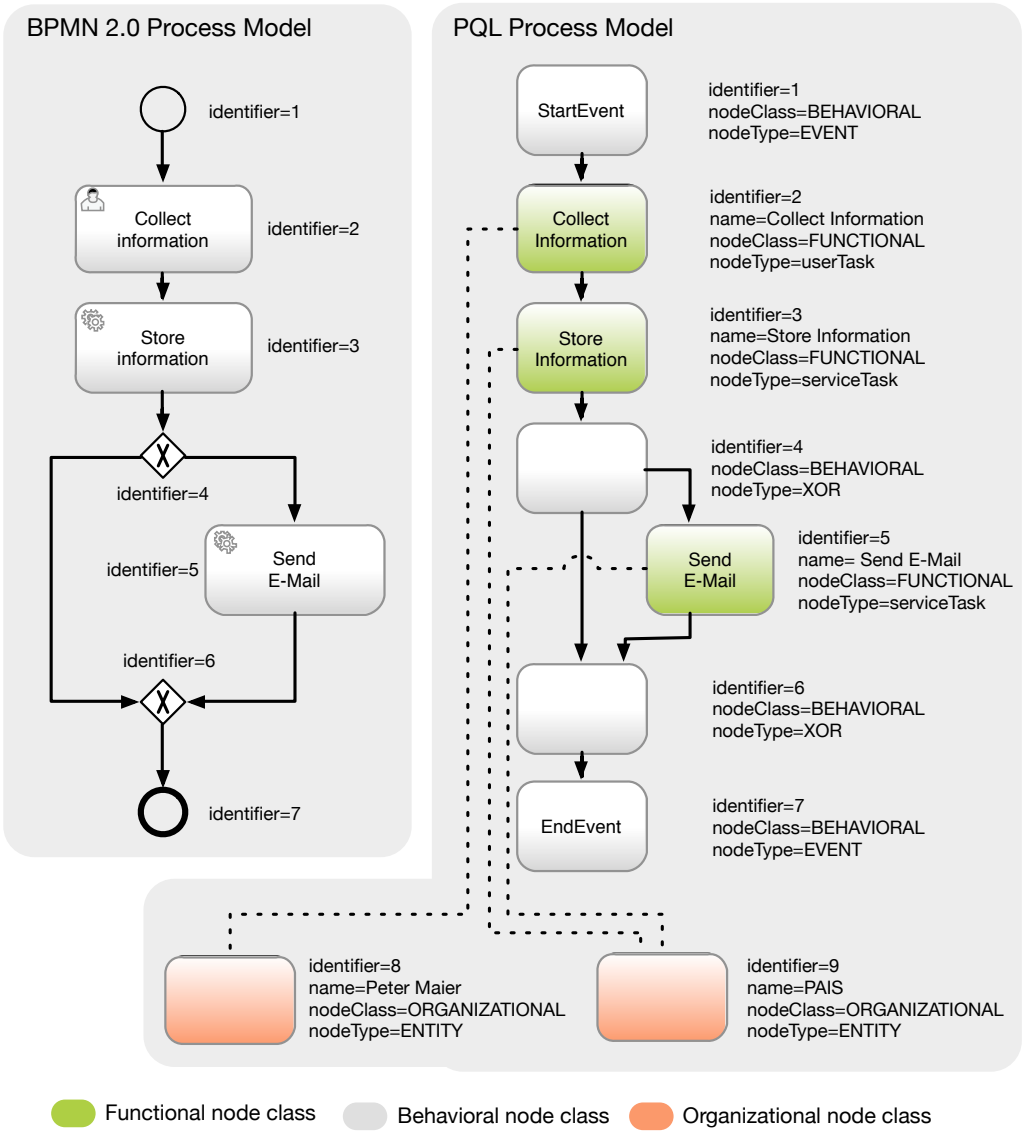

Figure 4.1: Mapping between a BPMN 2.0 Model and the PQL Process Model

different process models and between process elements. These dependencies may be also used to describe SESE blocks, where process model change operations may take place.

A *dependency* may exist between different process models, process fragments or process element attributes within process models. A process fragment is a sub-graph of a process

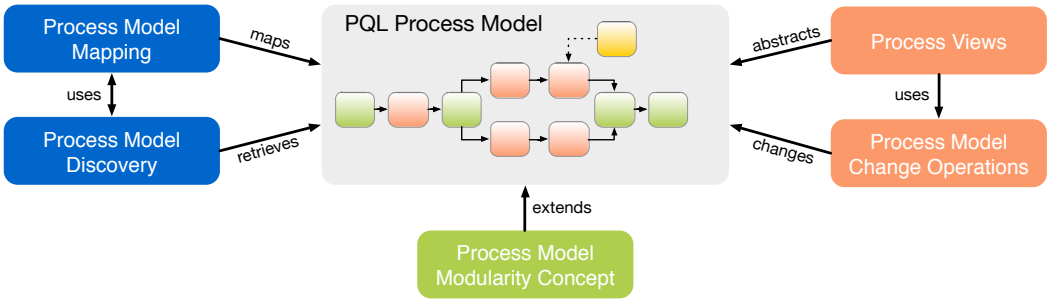

Figure 4.2: Overview on PQL Concepts

model and consists of a SESE block comprising a single node, a set of process nodes or a complete process model. Process nodes and edges in a process fragment contain attributes to differentiate between different process perspectives (cf. Section 4.1).

In order to discover process models in a process repository (cf. Section 2.4), it is necessary to outline information, which the desired process model has to contain. This information is either concerning the graph of a process model (i.e., *structural perspective*), or properties of process elements, like a name or identifier (i.e., *informational* or *functional perspective*).

Discovery based on a *structural perspective* may be described by defining process fragments and searching for exact matches between these process fragments and other process models. In practice, process models often contain process fragments consisting of tasks, which describe a domain functionality, but distinguish themselves by minor changes. For example, tasks within delivery departments have to handle containers and parcels in a different way, but basically perform the same abstract tasks: picking, packing, and dispatching products. Therefore, it is reasonable to define a discovery based on process model graphs by similarities, rather than just on exact matches.

Searching for process model similarities is a wide research field, in which different approaches exist to define such similarities. In [50] algorithms to describe similarities by calculating a set of necessary atomic process model change operations to convert a process fragment into another fragment are proposed. The fewer process model change operations are necessary, the higher is the similarity between these two process models. This approach takes similarities on a structural perspective as well as on other perspectives of a process

model into account (e.g., a similarity between two tasks with different name attributes can be described using a process model change operation renaming tasks).

Other approaches exist relying on *edit distances* [22, 58]. Edit distance is a measurement to quantify the similarity between two objects (e.g., strings or trees) by counting the minimum number of operations required to transform one object into another. There exist different definitions of edit distances [54, 83]. The *Levenshtein distance* between the strings "mark" and "parc", for example, is 2, because two substitution operations (i.e., from "mark" to "park" and "park" to "parc") are necessary. Process model similarity discovery is a far complex topic and cannot be discussed in detail within this thesis.

In order to enable discovering of process models by PQL, similarities on a structural perspective of a process model are described by process fragments. Therefore, a PQL process fragment based on the PQL meta-model uses an additional edge type indicating, that two process nodes are not direct neighbors and, thus, other nodes may exist in between the two nodes. Similarity between a PQL process fragment and a real process model is described by different metrics, in order to differentiate between *exact matching* and *similarity matching*. One example for such a metric could be the above mentioned edit distance realizing similarity matching.

Exact matching of process fragments is necessary to describe process model changes unambiguously. Therefore, the discovered process fragment contains the exact area on which process model change operations are applied. The next section illustrates these operations.

In order to outline a set of process nodes across a process model, that are not necessarily grouped together, comparisons on an attribute level are used. This type of dependency representation is necessary for PQL to support abstract process view descriptions (cf. Section 4.2.3), which do not necessarily need coherent fragments, but sets of process nodes to be processed (i.e., a node set of a view creation operation).

4.2.2 PQL Process Model Change Operations

Process model change operations defined in PQL have to be implemented supporting arbitrary process modeling notations (cf. Requirements GRQ-2 and GRQ-4). Hence, it is necessary to define a standard set of process model change operations, that are applicable on every process model notation [81]. A process model is initially transformed to a meta-model (cf. Section 4.1.2). Thus, all change operations do not modify a process model directly, but its meta-model representation. PQL provides a standard set of process model change operations, that on the one hand offer a best possible coverage of process model notation change cardinality, on the other hand still remain notation independent.

Standard process model change operations do not take the defined behavior of a process model into account (e.g., in order to define a process participant designated to execute a tasks two process elements have to be added to a process model: an ORGANIZATIONAL node representing the process participant and a LINK edge to describe the affiliation between both). Instead, they just modify single nodes and edges separately and, thus, are defined as atomic change operation. In order to support further process model change operations, for example to define a process participant as designated to execute tasks, these process model change operations may be combined to compound operations. Some compound operations, like moving sets of tasks in a process model, should be also implemented in order to simplify handling of change operations.

The standard set of process model change operations should include operations to change all described perspectives in Section 4.1 (cf. Requirement MRQ-3). *Behavioral change operations* modify the control flow of a process model. They can insert, modify, move, or delete process nodes and change attributes of process elements. *Functional change operations* are highly implementation-dependent and cannot be described further (e.g., functions in Event-driven Process Chains correspond to a variety of task types in BPMN 2.0, for example service and script tasks [74, 59]). They have to be defined notation and implementation-specific as a modular concept instead to be able to extend PQLs process model notation support (cf. Section 4.2.4). *Organizational change operations* are able to define process participants and their relation to tasks. *Informational change operations* insert, delete, and modify informational nodes.

Again, a separation between organizational and informational related nodes is necessary, because algorithms must be able to determine, whether process elements describe a certain perspective or not. Process abstraction algorithms, for example, must be able to decide, if a functional node has to be abstracted based on a given set of organizational constraints (e.g., *reduce all tasks, that are executed by a specific user*).

Behavioral change operations need a process fragment of a process model, which should be modified. In case a process model contains no gateways and thus no branches, an operation to insert a new node could be defined by the tuple $(X, successor(X))$. Therefore, node X references a node in the process model. Otherwise, when inserting more than one node not in sequence (i.e., surrounding nodes with two gateways) or insertions around control flow splits, every insert position has to be defined by a fragment consisting of exactly two process nodes. Alternatively, just one exact identifier attribute (i.e., definition of a specific node) has to be supplied as well as a parameter to define, whether a change operation should take place prior or after the defined node (cf. Figure 4.3).

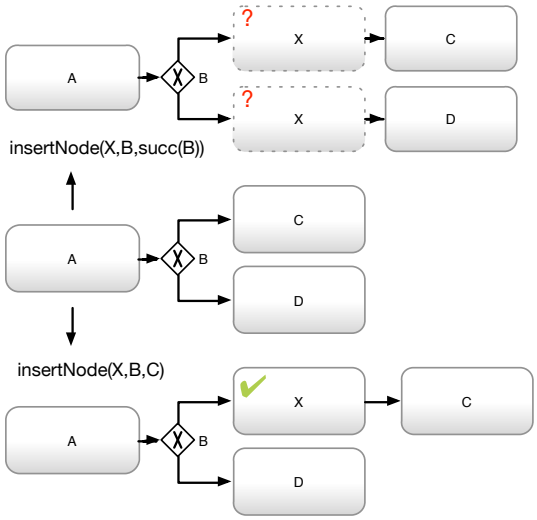

Figure 4.3: Ambiguities Occuring During Node Insertion

A limitation of this behavior is, for example, an insertion of a node after a gateway, which has more than one outgoing branch. Particularly, a specific branch is not defined and

the operation may directly affect any of the outgoing branches. Hence, a distinct process fragment definition should be preferred. The goal of PQL is to support change operations based on dynamically discovered process models. Therefore, ambiguity solving parameter sets can be used as described in the context of process view updates (cf. Section 3.1.4).

4.2.3 PQL Process Views

PQL should allow for creating process views. Therefore, a description on how to create process views out of CPMs is necessary.

The approach presented in section 3 describes the creation of process views by applying a creation set on a CPM. A creation set includes view creation operations, which either reduce or aggregate a set of nodes. Node sets are dependent from a specific CPM, because they may only contain process nodes uniquely present in a CPM. As a consequence, creation sets describing a process view are not applicable on arbitrary CPMs and, thus, have to be defined discretely for each CPM.

For certain use cases it is preferable to define an *abstract process view definition*. *Show only a process participants tasks* is an example for an *abstract process view definition*: process nodes affected by view creation operations are not chosen based on dedicated node sets, but on a process elements attributes (e.g., the process element attribute defining the process participant designated to execute a task). The latter may contain multiple conditions (e.g., *show only a process participants tasks, and, additionally, show all tasks automatically executed by a PAIS as aggregated process nodes*).

Hence, abstract process view definitions in PQL are based on an *abstract creation set (ACS)*. An ACS consists of tuples $T = (Priority, Condition, ViewCreationOperation)$ comprising an application priority $Priority$, a view creation condition $Condition$, and a view creation operation $ViewCreationOperation$ (cf. Section 3.1.2).

Process nodes affected by a view creation operation are assessed based on a *view creation condition* valid for each CPM. *View creation conditions* are logic conditions identifying sets of nodes based on process element attributes (e.g., on a process element type, i.e., service

Process Element Attribute	Description	Applicable BPMN 2.0 Process Elements	Example Value
Identifier	Identifier to recognize a process node. Every identifier is unique in every process model	All	identifier="5"
Name	Name of a process element shown to a user	All	name="Retrieve message"
Process element type	Element type of a process node	All	processElementType= user task
Candidate organizational entity	Organizational entity (i.e., users, roles, organizational units, or organizations) designated to execute a task	Tasks	candidateEntity= peterMueller
Gateway direction	Direction of a gateway	Gateways	gatewayDirection= converging

Table 4.3: Valid Process Element Attributes for View Creation Conditions

tasks, or user tasks). Table 4.3 shows process element attributes, a view creation condition may rest upon (cf. Section 2.2.2).

A view creation condition compares a process element attribute with a dedicated value in a predicate-like manner. Example *show only a process participants tasks* consists of one view creation condition validating, if a process node attribute *candidate organizational entity* defines the process participant as execution candidate for a process node.

Hence, view creation conditions have to be interpreted depending on the CPM on which an abstract process view definition should be applied on—meaning, that a view creation condition is translated into a node set, which, in turn, can be further processed by a view creation operation.

Example *show only my tasks and, additionally, aggregate all tasks executed by a PAIS* for process participant *Julia* illustrated in Figure 4.4 contains two view creation conditions combined by a logical AND operator. The first view creation condition defines, that all process nodes are reduced, whose process element attribute *candidate organizational entity* does not contain Julia (c.f. Figure 4.4a). The second view creation condition expresses, that all process nodes with process element type *service task* are aggregated (c.f. Figure 4.4b). The application of view creation condition a) prior b) results in process view c) and differs from the result, when applying b) prior to a) (c.f. Figure 4.4cd).

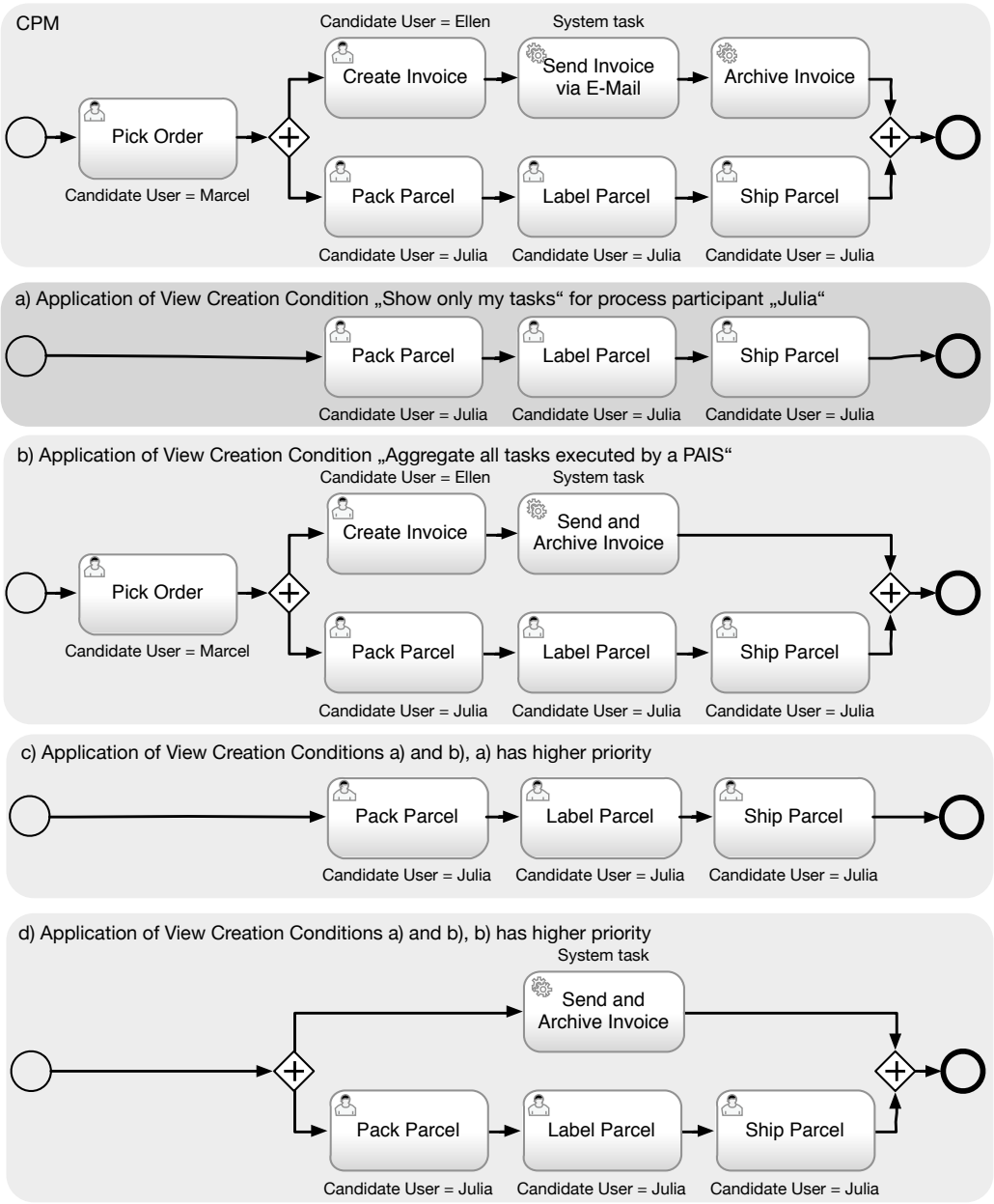

Figure 4.4: Application Priority for Abstract Process Views

Hence, view creation operations defined in an abstract process view definition have to be ordered by an *application priority*. Application priorities are necessary to solve arising intersections between different view creation operation node sets. Intersections of node sets may occur, because view creation conditions may affect the same process nodes and, thus, their node sets are not disjunct (c.f. Section 3.1.4). Therefore, the node sets of an applied view creation operation has to be excluded for all further view creation operations. In other words, process nodes already part of a view creation operation's node set are not considered for further view creation conditions to be applied on a process view.

Additionally, a correct application of aggregation operations has to ensured. Aggregations may only be applied on SESE blocks. As view creation conditions may define sets of nodes, that are not coherent in their control flow, an application of aggregations has to be ensured by first splitting node sets identified by a view creation condition into valid SESE blocks with the $minimalSESE$ algorithm (cf. Section 2.2.3). For example, if the CPM in Figure 4.4 contains an additional service task "Collect Information" before user task "Create Invoice", an application of view creation condition *aggregate all tasks executed by a PAIS* may only be executed, if the additional service task is excluded from the aggregation node set consisting of service task "Send Invoice via E-Mail" and service task "Archive Invoice". As a result, view creation condition *aggregate all tasks executed by a PAIS* is split up into two aggregation operations with node sets $N_1 = \{"CollectInformation"\}$ and $N_2 = \{"SendInvoice", "ArchiveInvoice"\}$.

4.2.4 PQL Modularity Concept

PQL has to be built modular to support various process model notations and PAIS implementation-specific constructs, like conditions for conditional branches. All operations and algorithms supplied by PQL are organized hierarchical. Therefore, PQL operations are divided into *notation-independent* and *notation-dependent* operations. *Notation-independent operations* only require information provided by the meta-model without any extensions to attributes (for example, node insertions). *Notation-dependent operations* are built as compound operations on top of notation-independent operations. For instance, a BPMN 2.0-specific operation to insert a service task first inserts a functional node and then inserts an additional attribute, which describes the functional node as service task.

Additional support for other process model notations is achieved by adding a module, that consists of a transformation description (cf. Section 4.1.2), definitions for implementation-specific constructs, change and abstraction operations, that are necessary to achieve a notations complete cardinality.

4.3 Software Architecture Supporting PQL

This section shows an architectural proposal for PQL, which meets all requirements presented in Section 3.3. First, *PQL requests* are explained. Then, all components of the software architecture supporting PQL are illustrated on a technical prospect. Lateron, the *PQL processing pipeline* with different processing steps is introduced. In this Section operations to be executed by the proposed software architecture are described as PQL requests intentionally. A PQL request is treated as protocol-like interaction between a client component requesting PQL actions and the PQL software component dispatching the latter.

Figure 4.5 shows, how a PQL request is being processed (cf. Section 4.3.3):

1. A user sends a PQL request (cf. Section 4.3.1).

2. A discovery component searches for process models requested in the PQL request (cf. Section 4.2.1)

3. Process model change operations, defined in the PQL request, are applied on a discovered process model (cf. Section 4.2.2)

4. Additionally, view creation operations are applied on a process model (cf. Section 4.2.3)

5. Additionally, a process model is transformed into different exchange formats, like XML or JSON

6. The final process model—or process view, if view creation operations are applied—is sent to the user

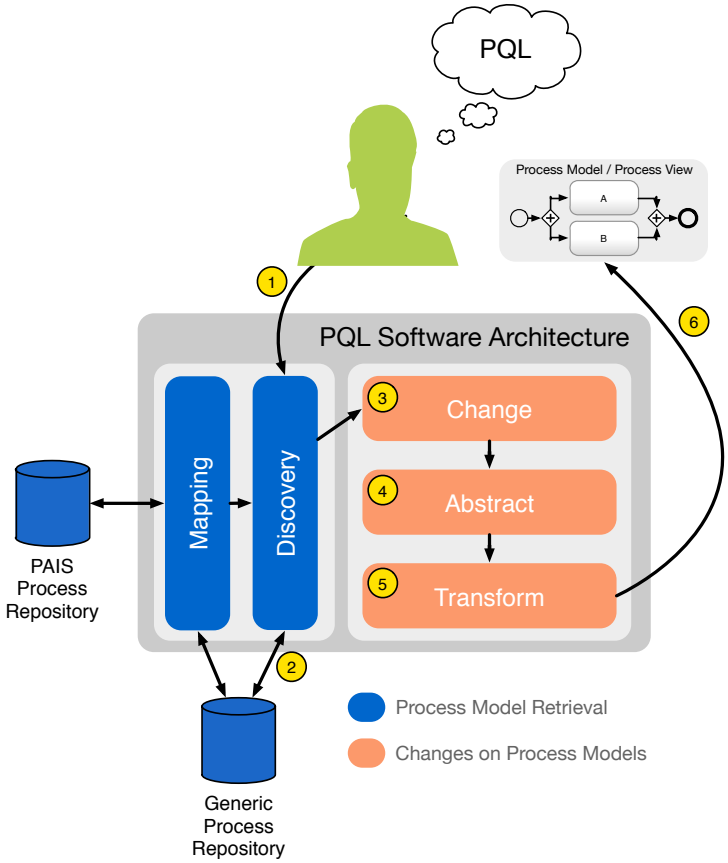

Figure 4.5: Overview on PQL Processing

4.3.1 PQL Request

A PQL request as part of a request-reply[2] is a message sent by a user or system containing a PQL string. The latter first has to be converted into an *intermediate representation* by a *parser*. Further, a parser is a computer program, that analyzes a string in order to associate strings with syntactic units of a grammar (i.e., a pre-defined rule-set), and transforms these groups into an intermediate, machine-readable representation.

A PQL request consists of the following parts: *Definitions*, *Discovery*, *Actions*, and *Transformations* (cf. Figure 4.6). Process models and process fragments are defined in part *definitions*. A discovery of process models may be manipulated by part *discovery*. Further-

[2]Message Exchange Pattern [35]

more, part *actions* defines process model change operations, view creation operations and the persistence behavior. Finally, transformations of a process model's exchange format may be defined in part *transformations*.

The PQL request is denoted as internal DSL based on the *Extensible Markup Language (XML)*. XML is a widely supported, *open standard markup language*, and thus first choice for PQL [25].

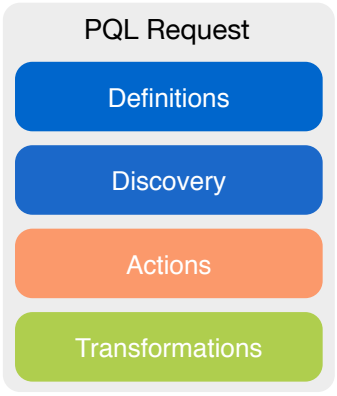

Figure 4.6: PQL Request

This section gives an example of a PQL request. Listing 4.1 shows a PQL request, where three fragments are specified (inside tag *definition*). Process fragments *fragment1* and *fragment2* are used to find two process models, where the two process fragments are present with a similarity value of 2 (fictional value for illustration) and process fragment *fragment1* occurs before process fragment *fragment2*. The resulting process models are transformed into a JSON-container respectively retrieved as PQL meta-model. The third process fragment defines the position and reference to the process model with id=specificModel1, where the fourth process fragment *fragInsert* is inserted. The resulting process model is stored in its origin process repository and updates the existing process model. Note, that the store-action defines an attribute with another namespace *activiti*. Changing the namespace and adding additional attributes for actions is one way to extend actions by implementation-specific constructs.

```
<pql>
  <errorhandler stage="modification" type="omit"/>
  <definition>
    <fragment refId="fragment1">
      <node id="user" type="userTask"/>
      <node id="script" type="scriptTask"/>
      <edge class="flow" srcId="user" trgId="script"/>
    </fragment>
    <fragment id="fragment2">
      <node id="script1" type="scriptTask"/>
      <node id="script2" type="scriptTask" name="generate thesis"/>
      <edge class="flow" srcId="one" trgId="endNode"/>
    </fragment>
    <model id="specificModel1">
      <node id="one" type="userTask"/>
      <node id="end" type="endEvent"/>
```

```
17        <edge class="flow" srcId="one" trgId="endNode"/>
      </model>
19      <fragment id="fragInsert">
          <node id="newNode" class="functional"/>
21      </fragment>
    </definition>
23
    <discovery>
25      <discover type="similarity" dId="discoverFragment1">
          <condition type="changePatternSimilarity" value="2"/>
27          <rel refId="fragment1"/>
          </condition>
29      </discover>
        <discover type="similarity" dId="discoverFragment12proximity">
31        <condition type="fragmentSimilarity" value="3">
            <rel predRef="fragment1" succRef="fragment2"/>
33        </condition>
        </discover>
35  </discovery>

37  <actions>
      <action type="search" aId="searchfragment1similarity" dId="discoverFragment1"/>
39    <action type="search" aId="searchfragment12proximity" dId="discoverfragment12proximity"/>
      <action type="change" aId="insertAction" refId="specificModel1">
41      <insertFragment refId="fragInsert"/>
      </action>
43    <action type="store" option="updateExisting" aId="insertAction"
          activiti:repository="repo1"/>
45  </actions>
    <transformations>
47    <transform aId="searchfragment1similarity">
        <format type="json"/>
49    </transform>
      <transform aId="searchfragment12proximity">
51      <notation type="meta-model"/>
      </transform>
53  </transformations>
</pql>
```

Listing 4.1: Example of a PQL Request

4.3.2 Architectural Components

In order to implement PQL different different components are necessary. The latter are organized by functionality and are required for the processing steps introduced in Section 4.3.3, for example, to transform process models into PQL process models and vice versa.

The *PQL controller* is the core component of a PQL processing software architecture. It handles calls between other components, manages the processing pipeline, and all persistence connectors. It further allows for registering new process model notations and custom operations to the notation transformation component and graph library.

Next, the *interpreter* is able to process a PQL request. Therefore, the latter has to be converted into an intermediate representation. The interpreter checks the PQL syntax against a PQL grammar, and parse tree (as intermediate representation) is generated. This parse tree consists of process fragment and dependency descriptions, references to process models stored in a process repository, and actions to influence the processing pipeline.

A *persistence connector component* retrieves and stores process models in a process repository. However, the process repository is not part of the PQL framework. Each persistence connector is PAIS-dependent and connects the PQL processing software architecture to a process repository. Pre-defined interfaces are required to be implemented, in order that all methods to retrieve and store process models are available to the PQL controller.

Process models retrieved from the persistence connector are described in a specific notation. The *notation conversion component* converts such process models with the help of so-called conversion descriptions into PQL process models. Process models and conversion descriptions can be handled by the conversion component to convert a process model into a PQL meta-model.

PQL's *dependency engine* manages dependencies between process fragments or process models and change operations to be executed by PQL. Dependencies and change operations are defined in a PQL request and can contain discovery operations to find a process model, change operations or abstraction operation. To be able to discover process models, the dependency engine is able to access respective process repositories.

Discovering process models by similarity search massively utilizes CPU and memory resources. As a result, all process models should be cached and optimized by the dependency engine. A possible optimization could be process element indexing, where process elements are organized by attributes or element types for a faster lookup.

The *graph function component* offers common graph algorithms, like calculating preceding or succeeding nodes and SESE fragments, checks for cycles or calculation of paths in

process models. These functions are provided for other components, or definitions within PQL requests.

The *process model modification component* changes PQL process models based on change operations, which are defined in a PQL request. It manages the structure, considers the expressiveness level and constraints of process models and ensures, that all change operations are applied correctly.

4.3.3 Processing Pipeline

The *processing pipeline* of a PQL request is similar to the data state model [16].

The latter describes data transformation steps required to visualize data. It is divided into stages and processing steps (cf. Figure 4.7). Stages show a status of processed data, while processing steps describe transformations of processed data. Steps act as transition between stages. The data state model describes four stages: *value* (raw data), *analytical abstraction* (meta-data, preprocessed data structure), *visualization abstraction* (visualizable information) and *view* (final visualization presented to a user).

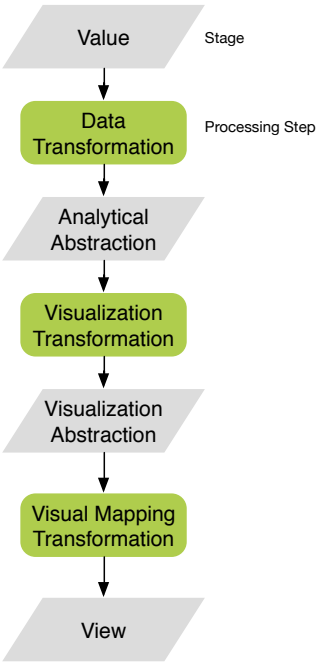

Figure 4.7: Data State Model

Between these stages there are three processing steps: *Data transformation* converts raw data into a preprocessed data structure, *visualization transformation* processes this data to visualizable information and *visual mapping transformation* takes this visualizable information and presents a graphical view. Thus, every output created by a user interface can be described with the data state model.

Every PQL request has to pass the processing pipeline, in which actions defined in the PQL request are processed (e.g., process model change operations). The processing pipeline consists of five processing steps: *discovery*, *modification*, *persistence*, *abstraction* and

retrieval performed in different components. The behavior of every step can be adjusted (i.e., some steps may be executed optionally). Figure 4.8 shows the PQL processing pipeline.

Step *discovery* transforms a PQL request into an intermediate representation required to be interpretable by the PQL software architecture and searches for process models in associated process repositories. To be able to search for process fragments, every process model is transformed into the PQL meta-model and then matched by the dependency engine.

If process model changes are defined in a PQL request, the *modification* step executes change operations on the PQL meta-model. A dependency engine is required to determine relevant positions for changes defined by process fragments (in the PQL request).

Step *persistence* transforms a changed PQL process model back to its original process model notation and stores it in a process repository.

After discovering a process model and optionally performing changes additional abstraction transformations can be executed. Therefore, view creation operations can be executed in the *abstraction* step (cf. Section 3.1.2).

Finally, the PQL meta-model is transformed back into its original process model notation and converted into a respective exchange format for process models, like XML or JSON.

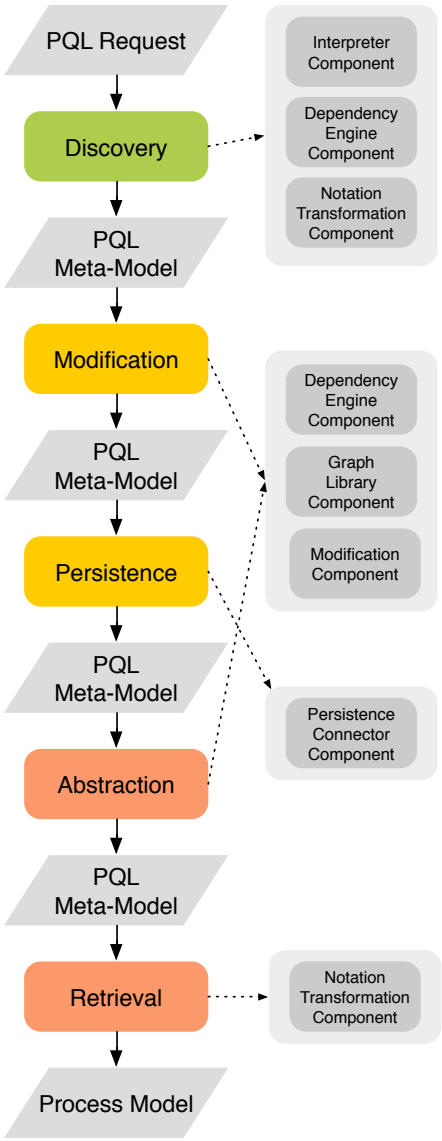

Figure 4.8: PQL Processing Pipeline

4.4 Summary

This section introduced a theoretical approach for PQL based on requirements outlined in Section 3.3). PQL is a DSL allowing for implementation-independent definition and management of process models and process views. PQL consists of a meta-model incorporating different process perspectives, change and process view creation functionalities altering the PQL process model, modules to enable process notation-specific meta-model extensions and operations, a dependency engine mapping relations between process models or process fragments and a concept to map different process notations through the PQL meta-model. The processing pipeline of PQL defines how PQL requests are handled.

The next section establishes the Activiti BPM platform and selected functionalities being the basis for the Clavii BPM platform proof-of-concept implementing a PQL subset. Clavii, in turn, is introduced afterwards.

5

Activiti BPM Platform

This Section introduces the *Activiti BPM Platform* (Activiti for short) used by the BPM platform developed as part of this thesis (cf. Section 6).

Activiti is an open source, Java-based PAIS released under the Apache License 2.0 [67]. It requires a Java Application Server (e.g., Apache Tomcat or JBoss). Activiti can be installed as *standalone* or as *embedded version*. The *standalone version* consists of the complete *Activiti Toolstack* (cf. Section 5.1), while the *embedded version* only contains the *Activiti Engine*. The latter version maybe required for implementing into existing Java projects (cf. Section 4). Activiti is very flexible and easy to integrate in various frameworks.

Section 5.1 gives an overview of all Activiti components, Section 5.2 introduces Activiti functionalities used by the Clavii BPM platform. Finally, Section 5.3 describes the *Activiti Application Programming Interface (API)* and presents some code examples.

5.1 Activiti Toolstack

The Activiti BPM platform consists of several components. The *Activiti Engine* is the core component. It provides process execution functionalities, like executing BPMN 2.0 process models and instantiating tasks.

Component *Activiti Modeler* is a web-based modeling environment to create BPMN 2.0 process models. It is based on an early version of the *Signavio Process Editor* component [67].

Component *Activiti Designer* can be used to enrich tasks of process models by executable code. It is based on Eclipse and offers widespread functionalities to develop PAIS-executable task components. Furthermore, it offers an integrated process modeling tool.

Finally, component *Activiti Explorer* (cf. Figure 5.1) provides users access to the process repository and allows to control the Activiti Engine component. For example, it is possible to start new process instances, loading worklists of users, or monitor active process instances.

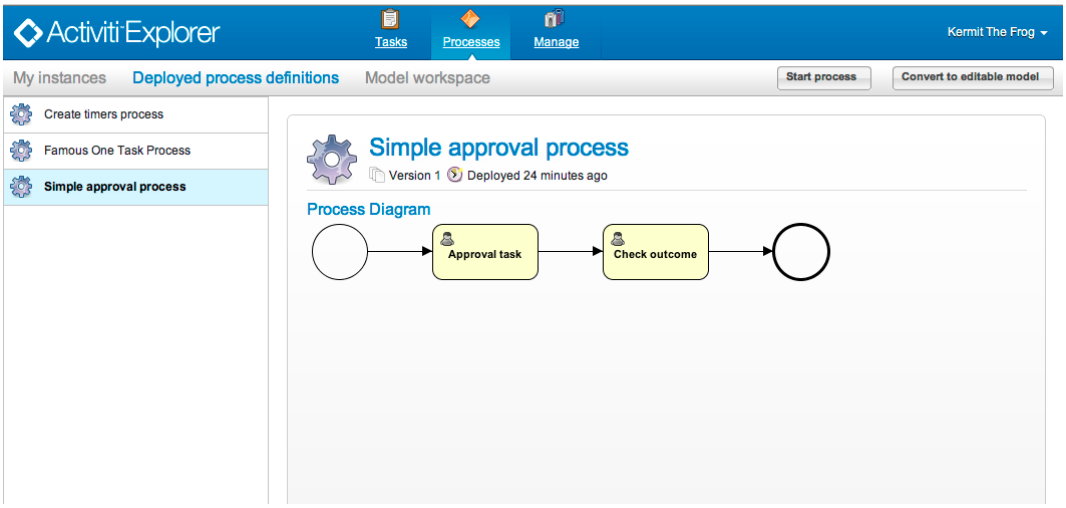

Figure 5.1: Activiti Explorer User Interface

Furthermore, a REST[1]-based [28] service API offers methods to manage functionalities of the Activiti Engine. In general Activiti Engine uses a H2[2] database [1], but other databases are supported as well. The following data is stored in a database:

- Users and groups for organizational models
- Process models
- Historic process instance data

Historic process instance data comprise, for example, execution time, execution results, and execution states (e.g., a task was executed correctly or failed during execution) of tasks, or values of process data elements.

5.2 Process Modeling Support

This Section explains important concepts used in this thesis, as Expressions and Custom Extensions. For a comprehensive introduction into Activiti's architecture please consider [67].

5.2.1 Java Object Representation for Process Models

Activiti has defined its own Java representation of BPMN 2.0 process models, which is described in the following. Please note: Subsequently, Java classes are written in *italic* font style.

Class *ProcessDefinition* is a metadata container for *BpmnModel*. A *BpmnModel* class represents one executable BPMN 2.0 process model. Class*ProcessDefinition* comprises a *MainProcess* class and additional *Processes* classes, which are linked to MainProcess class by either a *SubProcess* class activity class or by OrchestrationObjects, like Swimlanes and Pools. Class *BpmnModel*, optionally, stores classes of types *Artifacts*, *ItemDefinitions*, *Pools*, *Lanes*, *Messages*, *Problems*, and additional graphic information to describe how a

[1] Representational State Transfer - a programming paradigm for web-applications
[2] Light-weight Java-based relational database

process model is layouted. A process model contained in *BpmnModel* class is represented by the *Process* object. It consists of a control flow elements set, attribute *ioSpecifications*, attribute *artifacts* and attribute definitions for users and groups, which are eligible to start a process model (i.e., candidateStarterUsers and candidateStarterGroups).

BPMN 2.0 elements introduced in Section 2.2.2 are designed as Java classes implementing interface *BaseElement*. Figure A.1 gives a short overview about the class hierarchy. Class *BaseElement* can be extended by an *ExtensionAttribute* class as described in Section 5.2.3. Furthermore, all control flow relevant process elements extend class *FlowElement*. Class *FlowElement* is divided into *SequenceFlows* and *FlowNodes* classes (i.e., *DataObjects* are not used by the BPM platform introduced in Section 6, and therefore not treated in the following). Thereby, a FlowNode represents either class*Event*, *Gateway*, or *Activity*, which are connected with *SequenceFlow* classes.

5.2.2 XML Representation for Process Models

Process models represented as Java object *ProcessDefinition* (cf. Section 5.2.1) may be converted to a corresponding XML-based representation. Particularly, Activiti provides direct conversion support.

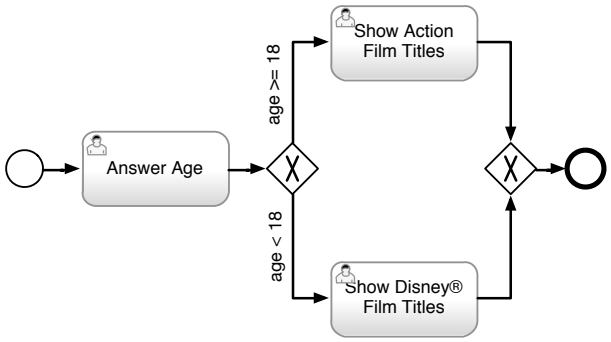

Figure 5.2: BPMN 2.0 Process Model for Age Verification

Listing 5.1 shows a XML-based BPMN 2.0-compliant process model for age verification (cf. Figure 5.2). Every process element consists of different attributes, like an id or a name (cf.

Section 2.2). The process model starts with a start event, followed by a user task. The latter defines a user form requesting the age of process participant *Fury*. The age is stored in a variable with id=age and is used to decide, which branch is activated. The first branch with id=splitflow1 is activated, if process participant *Fury* entered a number less than 18. In this particular case, user task with id=userTask3 only shows Disney® film titles. Otherwise, user task with id=userTask2 is activated showing action film titles to process participant *Fury*.

```
<definitions xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:activiti="http://activiti.org/bpmn"
  typeLanguage="http://www.w3.org/2001/XMLSchema"
  expressionLanguage="http://www.w3.org/1999/XPath"
  targetNamespace="http://www.activiti.org/test"
  xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL">
  <!-- root element -->
  <process id="ageVerification" name="Age verification" isExecutable="true">
    <startEvent id="startEvent"/>
    <sequenceFlow id="flow1" sourceRef="startEvent" targetRef="userTask1"/>

    <userTask id="userTask1" name="Answer Age" activiti:assignee="Fury">
        <extensionElements>
            <activiti:formProperty id="age" name="Insert age" type="integer"/>
        </extensionElements>
    </userTask>

    <sequenceFlow id="flow2" sourceRef="userTask1" targetRef="xorsplit"/>
    <exclusiveGateway id="xorsplit" default="splitFlow1"/>

    <sequenceFlow id="splitflow1" sourceRef="xorsplit" targetRef="userTask3">
        <conditionExpression xsi:type="tFormalExpression">
            <![CDATA[\${age < 18}]]>
        </conditionExpression>
    </sequenceFlow>

    <userTask id="userTask3" name="Show Disney Film Titles" activiti:assignee="Fury"/>
    <sequenceFlow id="joinflow1" sourceRef="userTask3" targetRef="xorjoin"/>

    <sequenceFlow id="splitflow2" sourceRef="xorsplit" targetRef="userTask2">
        <conditionExpression xsi:type="tFormalExpression">
            <![CDATA[\${age >= 18}]]>
        </conditionExpression>
    </sequenceFlow>

    <userTask id="userTask2" name="Show Action Film Titles" activiti:assignee="Fury"/>

    <sequenceFlow id="joinflow2" sourceRef="userTask2" targetRef="xorjoin"/>
    <exclusiveGateway id="xorjoin"/>
    <sequenceFlow id="flow3" sourceRef="xorjoin" targetRef="endEvent"/>
```

```
42      <endEvent id="endEvent"/>
    </process>
44 </definitions>
```

Listing 5.1: Activiti BPMN 2.0 XML Representation

5.2.3 Custom Extensions of Process Models

All BPMN 2.0 elements are enhanceable by custom extensions (cf. element userTask1, Line 12 in Listing 5.1). Extensions are necessary if, for instance, a graphical editor requires more attributes; e.g., coloring. The basic class for extensions is class *ExtensionElement*, which consists of one or more *ExtensionAttributes* classes. Class *ExtensionElement* can be added to every class, which implements the *HasExtensionAttributes* interface. A class *ExtensionAttribute* is a key-value object, similar to XML attributes within a XML tag.

5.2.4 Expression Language

Expressions in Activiti are used for service and script tasks, listeners, and conditional sequence flows. Conditional control flows, for example, must have the ability to map conditions in order to be evaluated by the Activiti Engine.

Particularly, Activiti supports the *Unified Expression Language (UEL)*, which is part of the Java JEE-specification [21]. To be more precise, it uses the *Java UEL Implementation (JUEL)*. Particularly, JUEL supports resolving Java Beans [71] and handles array, map, and list objects. UEL defines two types of expressions: *Value expressions* and *Method expressions*

Value expressions resolve to a String value. A value expression can contain variables, which represent *DataElements* classes associated to the process model or registered Spring[3] beans [46]. An example of a value expression is `${registeredBean.value}` or `${variable}`.

[3] open-source application framework for Java - http://www.spring.io/

Method expressions invoke a Java method to return a specific value, for example, `${registeredBean.getValues('order-9')}`. Method expressions in the Activitis Process Modeler component can be defined by free text.

5.3 Activiti Server Component and Java API

The Activiti server component running in a Java application container consists of different components (cf. Figure 5.3): the Spring component consists of a Spring container, expressions, and beans, whereas, the Activiti Engine component consists of an Activiti Java API, core services and the Process Virtual Machine (PVM). The PVM itself consists of a state machine model and a persistence layer, which communicates directly with underlying databases.

In order to develop a Java application embedding the Activiti Engine component, the *Activiti Java API* can be used. The latter offers methods to invoke methods provided by the Activiti Engine (i.e., starting a new process instance).

Hereinafter all core interfaces of the Activiti Java API are introduced.

Service *FormService* offers access to the Activiti form engine to define forms in process models. Service FormService is also responsible for rendering these forms in HTML.

Service *HistoryService* provides information and metrics about completed process instances. It exposes mainly querying capabilities.

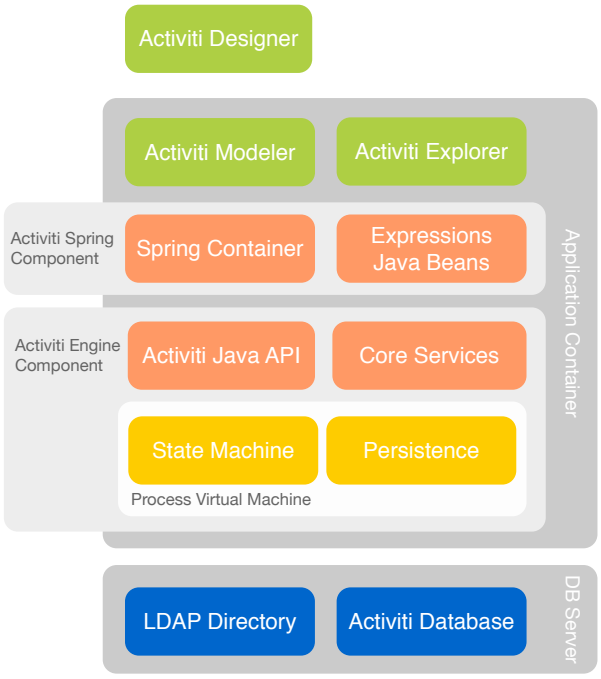

Figure 5.3: Activiti Architecture Overview

Service *IdentityService* is the authentication interface. Activiti handles authentications by itself, because a close coupling between all Activiti components is needed.

Service *ManagementService* provides methods to access Activiti database tables directly and execute asynchronous jobs. This is useful, if core service interfaces do not provide methods for advanced use cases. Jobs are used, for example for timers and asynchronous activity execution.

In order to deploy, query, delete and access process definitions, developers can use the service*RepositoryService*, which is also responsible for versioning of those entities.

Service *RuntimeService* is able to start and query process instances and access process data elements, which are defined in a process model (cf. Section 2.2.2).

Service *TaskService* represents a worklist manager. Using this service it is possible to create, claim, execute, and cancel user tasks.

5.4 Summary

The Activiti BPM Platform is an open source, Java-based PAIS providing numerous managing functionalities for process models and process instances. Activiti is divided into different components (i.e., the Activiti Engine and a REST-based API). The Activiti Engine, in turn, may be used as standalone component or may be embedded into other Java applications. Therefore, it offers the Activiti Java API containing various services (i.e. a repository service to be able to store process models or a run-time service to control process instance execution). Process models in Activiti are represented by a Java object model and may be transformed into a XML-based representation. Activiti process models may be extended by custom extensions (e.g., to add additional attributes to process elements). The next section introduces the Clavii BPM platform, utilizing technologies presented in this section.

6

The Clavii BPM Platform

Today, PAISs are very powerful and thus complex. In particular, they target at mid-sized to large companies and need strong knowledge of business process management. Small companies face BPM challenges with little or no BPM knowledge. Therefore, a PAIS should support them by offering advanced methods and concepts reducing efforts on building and managing a BPM-centric infrastructure.

As described in section 2.2, a BPM-centric infrastructure can be developed by following a *top-down approach* [29]. This means, that business processes are first documented on a high-level perspective, which result in coarse-grained process models. Subsequently, the latter are more detailed by fine-grained process models. Typically, process models are created by dedicated process designers. However, the top-down approach contradicts the aim of process views to tighten integrate process participants in the BPM lifecycle.

In contrast, a *bottom-up approach* is required, which enables every employee to document its own business processes by using a simple and easy to set up PAIS. Furthermore, they should be able to easily execute their PAIS. In particular, a BPM-centric IT architecture has not to be extensively planned.

Addressing these issues, we introduce the Clavii BPM platform (Clavii for short) in the following. Clavii is developed for small companies, that do not have a lot of BPM knowledge. In particular, Clavii should decrease the time to design and implement process models by showing up various concepts, for example, the PQL query language presented in Section 4.

Section 6.1 shows further aspects of the realization first, then the Clavii proof-of-concept implementation is explained. Section 6.2 illustrates the software architecture for Clavii. Section 6.3 explains functionalities provided by the proof-of-concept implementation. Section 6.4 describes, how process model management is implemented. Section 6.5 shows up the proof-of-concept software architecture for PQL implemented in Clavii. Finally, Section 6.6 explains creating and updating process views.

6.1 Principles

The Clavii BPM platform follows three main design goals: *simplicity*, *open standards*, and *modularization*.

Simplicity itself is a broadly defined term and maybe further categorized to: range of functions, handling, and presentation.

Range of Functions: The main purpose is to develop a PAIS that is tailored for non-technical users. Hence, it is reduced to common process elements. Block-structured process models are applied since they are easier to understand [56]. Next, the *correctness by construction principle*, similar to ADEPT, ensures control and control flow correctness at any time [68]. Advanced modeling constructs, like Events and synchronized concurrencies, are cut out not to overexert users with limited BPM experience [84]. Branching Conditions are also simplified: they can be defined in advance or decided at run-time.

Simplicity in Handling: As a web-based platform, no installation and configuration should be required at client-side. Next one user interface, should provide functionalities of the BPM lifecycle. Hence, each BPM lifecycle step can be done seamlessly without context switching (i.e., application). *Simplicity in Presentation*: The user interface should be intuitive to use [57]. Hence, no time consuming trainings for users are required. *Open source frameworks* are applied for development, for example, Hibernate [47] and Activiti (cf. Section 5) [67]. Configurations (e.g., for persistence handling), as well as process models, are stored in open XML-based documents. Third-party tasks may be implemented by plain Java POJO (i.e., Plain old Java object) classes, a store enables users to extend task capabilities, buy pre-developed process models for common purposes.

6.2 Proof-of-Concept Implementation Architecture

The Clavii proof-of-concept implementation architecture is built as integrated Java EE container [33]. Its architecture and interface definitions allows to separate and change each part of the platform. This can be done, for example, by moving the user interface logic to a different Java EE container. In particular, central server components do not have to be modified i.e., a clear separation between the data tier and business tier exists (cf. Figure 6.1).

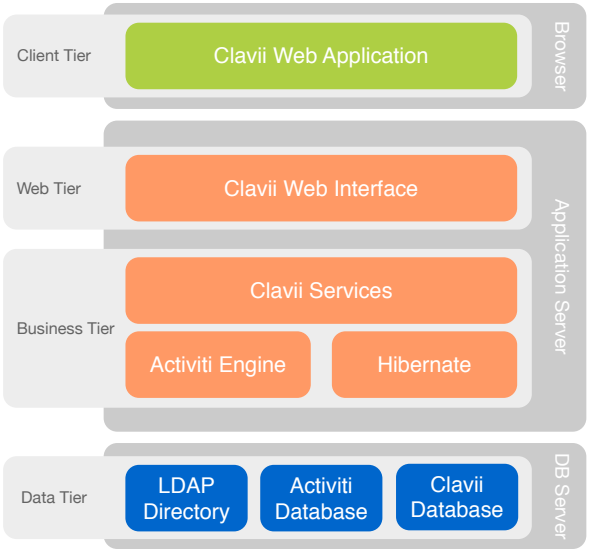

Figure 6.1: Clavii Architecture Overview

Clavii is mainly divided into a MVC[1]-like architecture. A model module defines common-used objects, like process models, user definitions or file container (cf. Section 6.2). The view module contains all logic related to render the user interface and dispatch user interactions (cf. Section 6.2). The business logic itself is defined in the controller module (cf. Section 6.2).

[1] Model-View-Controller architecture pattern

The Clavii *Model* module defines objects, like users, groups, process models, attachments, or settings [4]. These objects are persistable using the Hibernate framework [47], i.e., storing and retrieving information from the Clavii database is transparent. Therefore, the Data Access Objects (DAO) design pattern [61] provides an interface to abstract all Java objects from the persistence layer (cf. Figure 6.2). Each object extends class *DAO*, which offers access methods to change each entity.

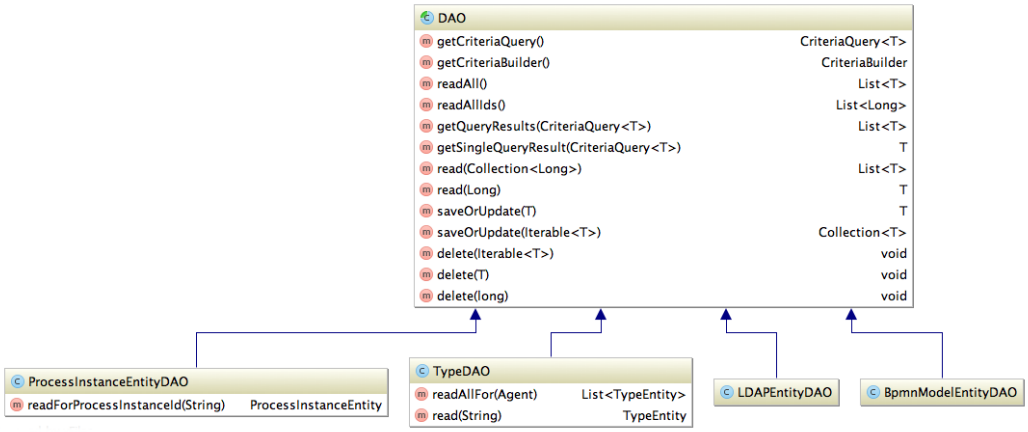

Figure 6.2: DAO UML Class Diagram Excerpt

The Clavii *View* module comprises the user interface (UI). The UI is based on Google GWT [53] and allows to access all functionalities with a single web-based application. Figure 6.3 shows the Clavii interface with a process model in BPMN 2.0. Clavii allows for rapid prototyping, every process model is executable from the start. Missing process data elements or decisions for branches are requested upon execution.

Further information about the Clavii UI is available in [15].

The *Controller* module holds all server-based functionalities: Figure 6.4) shows the Clavii controller architecture comprising packages for identity management, process instance monitoring, the plugin architecture, persistence handling, run-time management, process model and process view management (i.e., involving PQL, cf. Section 6.4, Section 4.3), and Section 6.6), and validation management shortly described in Section 6.3.

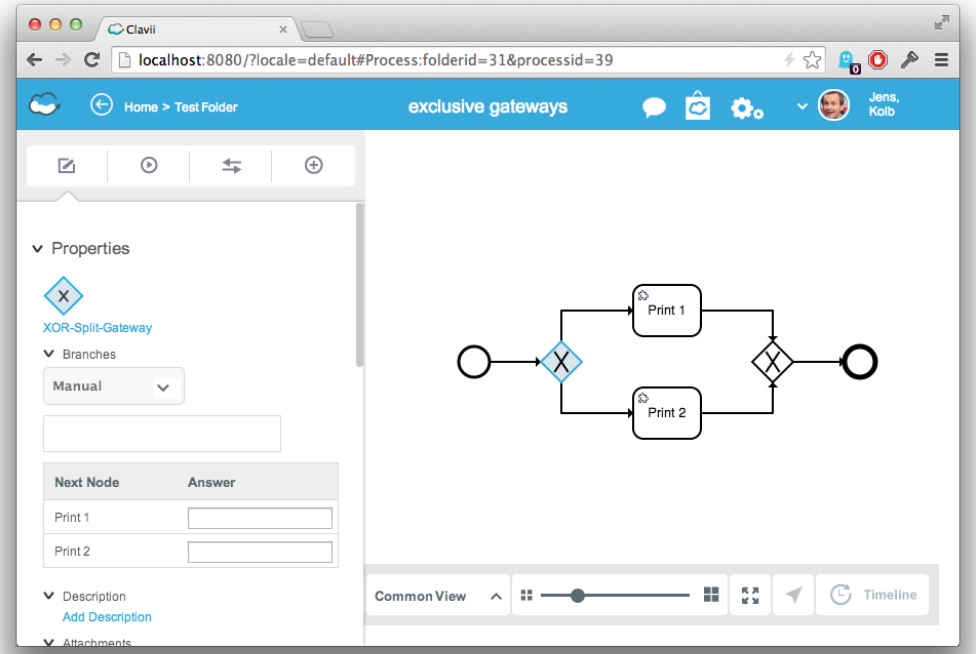

Figure 6.3: User Interface of the Clavii BPM Platform

6.3 Functionalities

Clavii is divided into different components. Each component provides functionalities for different domains and is represented by a *Manager* interface. Methods invoked on a *Manager* interface are transaction-enabled, i.e., they may be automatically reverted, when Java exceptions occured during execution.

Identity Management: Clavii offers *identity management* functionalities to identify and authenticate users represented by two different services: the *AuthenticationManager* service and *OrgModelManager* service. The AuthenticationManager service is used to authorize a login, create, update, or delete an *Agent* (i.e., a user). OrgModelManager service offers methods to manage organizations, organizational units and user roles. Clavii has a

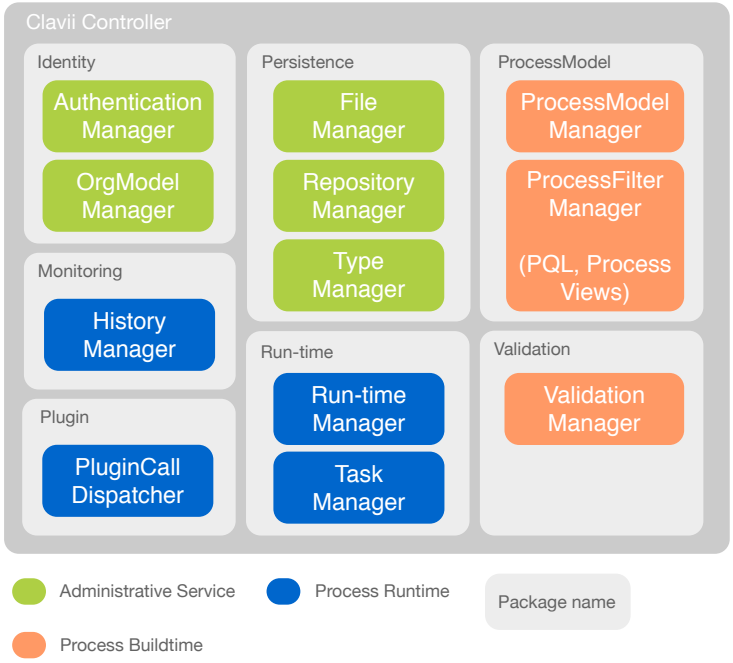

Figure 6.4: Clavii Controller Overview

built-in LDAP-connector and allows for synchronization with LDAP-schemes stored in a LDAP-directory.

Process Instance Monitoring: In order to recapitulate executed process instances, the *HistoryManager service* offers access to historic execution data. The latter is used to show finished process instances to users to check, if, for example, all containing tasks are executed successfully. Figure 6.5 shows a terminated process instance, where task "Print 1" was successfully executed, while task "Print 2" was not executed.

Persistence Handling: Clavii uses the RepositoryService of Activiti in order to store process models to be executed [67]. Persistence handling for process model related objects is encapsulated by the *RepositoryManager service*, while others, like attachments and icons are stored directly to Data Access Objects (DAO) [61]), or utilizing *FileManager service* and *TypeManager service* respectively.

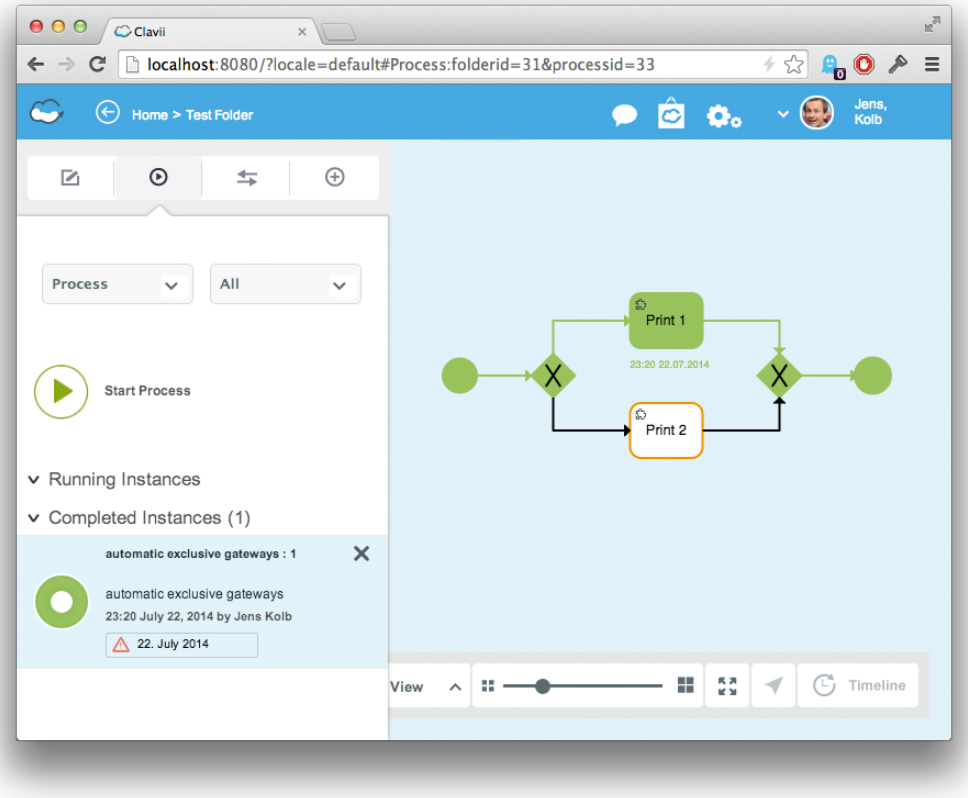

Figure 6.5: Process Instance Monitoring in Clavii

The *RepositoryManager service* implements methods for deploying, importing, accessing, updating, and deleting process models. If a process model is not deployed, which means that it can not be accessed and executed by the Activiti Engine, it is stored in a *BpmnModelEntity* object and persisted by a DAO. When a process model has to be executed, it is deployed to the Activiti RepositoryService by the *RuntimeManager* (cf. Section 6.3).

The RepositoryManager service also implements convenience operations, for example, to search for process models belonging to a specific organizational unit. These operations use a *CriteriaBuilder* to search for defined attributes.

Process Model Management in Clavii is implemented by two services: *ProcessModelManager* service and *ProcessFilterManager* service. The *ProcessModelManager* service offers methods to execute change operations on process models, while the *ProcessFilterManager* service implements methods to create and update process views (cf. Section 6.4) in conjunction with PQL (cf. Section 4).

Run-time Management in Clavii is divided into managers: the *RuntimeManager* and *TaskManager*. Service *RuntimeManager* is responsible to interact with the embedded Activiti Engine to enable the execution of process instances (cf. Section 5.3) . If a process model should be executed it is deployed to Activiti RepositoryService. Subsequently the RuntimeManager triggers the conversion of the process model to a process instance, which is then executed by the Activiti Engine. Service *TaskManager* provides methods to retrieve task lists of users and executes respectively modifies states of user tasks. The number of all untreated tasks, as well as untreated tasks for a specific process instance, can be fetched for every user. Both managers are used to separate business logic of Clavii from Activiti Engine. Hence, Activiti Engine can be easily exchanged by another BPM engine.

Validation Management in Clavii is implemented by service *ValidationManager* and offers methods to ensure a process model's correctness. The service may check conditional gateway expressions and PQL requests for their validity and returns a *CheckReport* object containing detailed information.

Plugin Architecture: An OSGI[2]-based *plugin architecture* allows for extending the number of available tasks, that may be executed by Clavii (i.e., ScriptTasks, ServiceTasks) [4]. Every plugin may be developed as POJO (i.e., Plain old Java object) and registered using XML-based service descriptions.

Data Type Framework: Usually, a process modeling notation offers process data elements to define a data flow between tasks. In Clavii, process data elements are not embedded into a process model, but managed by Clavii. Process data elements are built hierarchically [4]. Therefore, strong type handling ensures interoperability between service tasks and process data elements. The data type framework is easily configurable and expandable by end-users. Figure 6.6 shows a process data element definition for a *customer* consisting of five process data elements: "Name", "Age", "Birth Date", "Gender" and "Regular Customer".

[2]Component model based Service Delivery Platform [3]

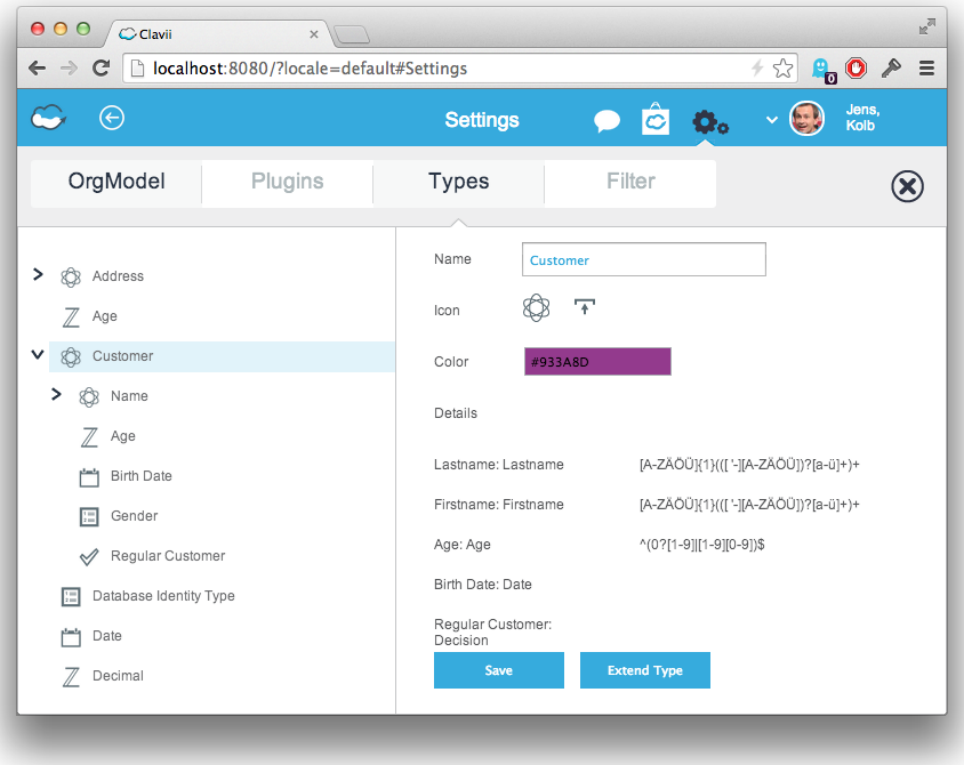

Figure 6.6: Process Data Elements in Clavii

6.4 Managing Process Models

Process model changes are handled by service *ProcessModelManager*. The latter only contains a few methods. Particularly, method `updateModel(ClaviiBpmnModel, ModelChangeDescription)` updates a process model based on an operation defined by class *ModelChangeDescription* (cf. Section 6.4.5). Every operation, in turn, is dispatched by a *ProcessModelDispatcher*, which routes it for processing to the service *ClaviiFlowFactory*. The latter offers process model change operations (cf. Section 6.4.5), which finally change the given process model.

In the following the process model representation is introduced (cf. Section 6.4.1), and its logical restrictions (cf. Section 6.4.2). Afterwards, the structure of change operations is introducted and its Clavii internal procedure (cf. Section 6.4.5). Finally, the process model filter execution is explained (cf. Section 6.6).

6.4.1 Process Model Representation

Process models in Clavii are represented by class `ClaviiBpmnModel` - a container, which includes an Activiti `BpmnModel`, a precalculated RPST (cf. Section 6.4.2) and a topological map. Figure 6.7 shows such a process model in the Clavii modeler. Topological ordering of a directed graph, or in this case a business process model, is a linear ordering of its vertices [31]. There exist redundant information between the RPST and the topological map, but this method saves run-time at minimal memory overhead for some graph operations. Aditionally *ClaviiBpmnModel* offers helper methods for debugging, the process model and the RPST for example can be converted into the DOT-format, which is a text-based graph representation used by many applications [72]. RPST and map generation are executed every time a process model is converted into a Clavii process model. A RPST of a process model can be computed in linear time, local changes in a process model only result in local changes of a RPST. Hence using a RPST for structural checks of a process model is very efficient, even if a process model is changed and an already computed RPST of the model has to be adapted.

In Clavii, it is possible to import existing process models in Activiti's XML-format (cf. Section 5.2.2). If block-structural ambiguities occur (cf. Section 6.4.2), it is possible to solve these by adding *BlockStructureExtensions* manually. BlockStructureExtensions are a subset of *ExtensionsElements*, which were developed to enhance the Activiti BpmnModel representation. Every *BaseElement* can be enhanced by an *ExtensionsElement*. In Clavii Extensions can be easily written by extending the generic *ExtensionManager* class. Figure 6.1 shows extensions for a conditional gateway. `clavii:properties` are a simple key-value storage and can be used by any Clavii component. `clavii:blockStructure` is automatically added to any gateway after importing an existing Activiti process model. Additionally every gateway altering change operation keeps the Extension consistent.

```
[..]
```

```
<process id="simpleProcess" name="Simple process" isExecutable="true">
[..]
<exclusiveGateway id="xorsplit" default="splitFlow1">
  <extensionElements>
    <!-- Extension for generic properties -->
    <clavii:properties xmlns:clavii="http://www.clavii.com/extensions">
      <clavii:property clavii:key="mode" clavii:value="manual"/>
    </clavii:properties>
    <!-- Extension for Block Structure -->
    <clavii:blockStructure
        xmlns:clavii="http://www.clavii.com/extensions">
      <clavii:correspondingId>exclusivegateway2</clavii:correspondingId>
      <clavii:seseType>ENTRY</clavii:seseType>
    </clavii:blockStructure>
  </extensionElements>
</exclusiveGateway>
[..]
```

Listing 6.1: Process Model Extension Elements

Listing 6.2 shows *ExtensionElements* automatically added, if a *ClaviiBpmnModel* represents a process view. Therefore, the name and internal ID of the applied process filter (cf. Section 6.6) is added, as well as the executed PQL-String. `clavii:nodeSet` references to *FlowElement* IDs, which were modified by the process filter engine.

```
[..]
<process id="simpleProcess" name="Simple process" isExecutable="true">
[..]
<extensionElements>
  <clavii:filter xmlns:clavii="http://www.clavii.com/extensions">
    <clavii:filterId>42</clavii:filterId>
    <clavii:filterName>User Tasks</clavii:filterName>
    <clavii:pqlString>GET MODEL bla</clavii:pqlString>
    <clavii:nodeSet>
      <clavii:reducedNodes>124,14,15,465,43,32</clavii:reducedNodes>
      <clavii:aggregatedNodes>34,11</clavii:aggregatedNodes>
    </clavii:nodeSet>
  </clavii:filter>
</extensionElements>
[..]
```

Listing 6.2: Process View Extension Elements

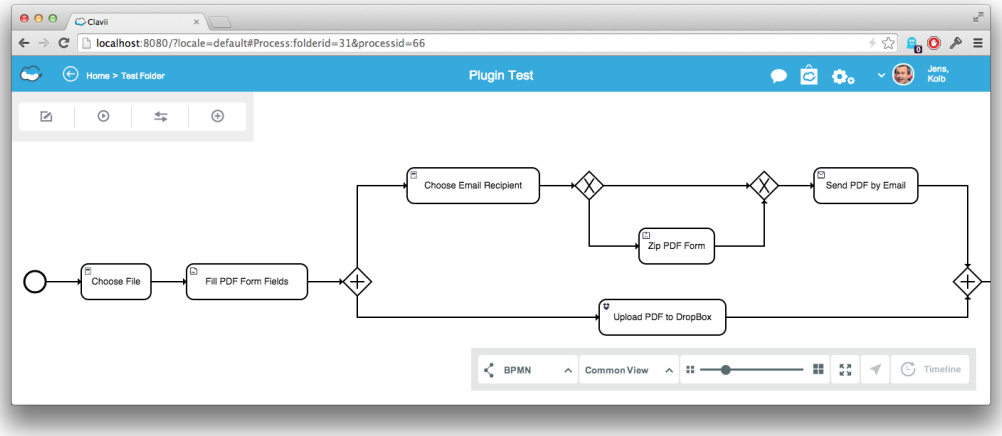

Figure 6.7: Clavii Process Model Excerpt

6.4.2 Block-Structural Constraints

Clavii is limited to block-structured process models enabling the use of process view creation algorithms [9, 43] and supporting users at build-time [18]. In general, structured process models are well defined by construction, easier to understand and allow for advanced modeling support.

To determine SESE fragments (cf. Section 2.2.3) used by the Clavii *GraphUtils* library, a RPST graph is used.

The RPST algorithm, that calculates a RPST graph, first decomposes process models into fragments. A fragment is a subset of the process model graph, here representing a set of edges. Every fragment has exactly one incoming and one outgoing edge. It is *canonical*: it does not overlap with other fragments, but can contain them. Fragments, again, are based on triconnected components, which have specific characteristics [77]. They can be obtained by further applying *split* operations.

Bond fragments consist of 2 nodes and $k \geq 2$ edges. A *Polygon* fragment is a graph with $k \geq 3$ nodes and k edges contained in a cycle. *Rigid* fragments can be further divided into

Polygons and Bonds, these again cannot be split further. A fragment is defined as *Trivial*, if it contains one single edge.

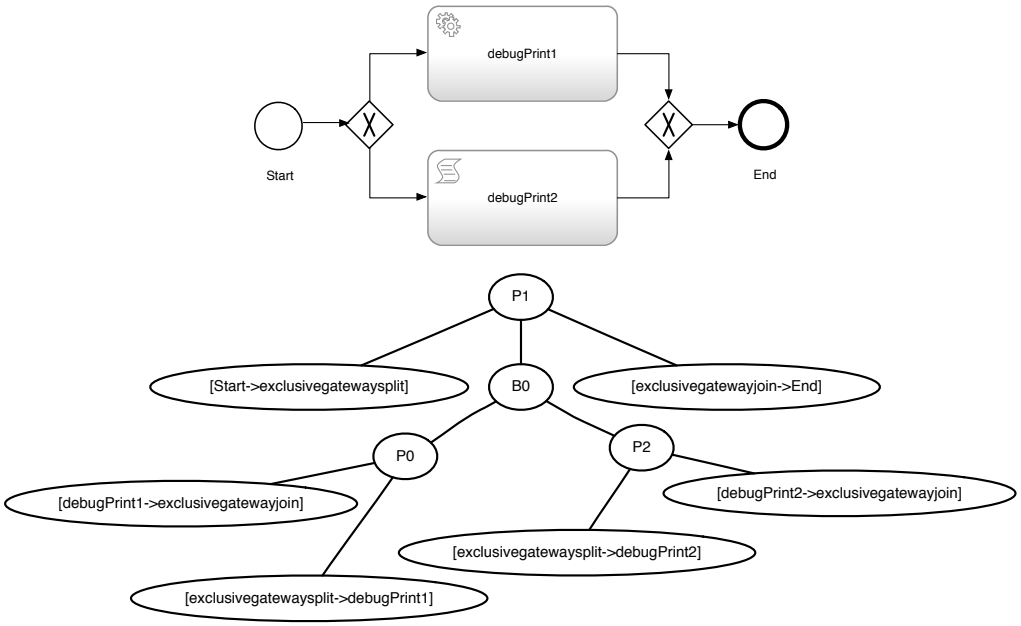

Figure 6.8: Process Model and Corresponding RPST Graph

A RPST is a set of these canonical fragments. It can be represented hierarchically as tree. The root node of a generated RPST tree represents the whole process model, whereas a leaf represents a *trivial* fragment. Figure 6.8 shows a BPMN 2.0 process model and its generated RPST graph. Nodes marked with *P* are polygon nodes, those node marked with *B* are bonds. *P1* is the root polygon and represents the whole process model. Note, that each branch of the gateway has its own polygon.

A simplified algorithm of the RPST generation shows as follows [66]:

1. G is a directed multi-graph
2. Compute a normalized version of G
3. Generate a tree T of the triconnected components of G

4. Remove all trivial fragments of T, which are not contained in G

5. Remove all redundant fragments in G

6. T is the RPST of G

Step 4 is necessary, because process nodes with more than one incoming and coincidently more than one outgoing edge (occurs, when a process model contains gateways) are split up into two nodes. These either have more than one incoming edge and exactly one outgoing edge, or vice versa. Additionally, virtual nodes were added, for example between the start node and the end node, which then have to be removed.

Every generated RPST can differ due to the fact, that a process model is described by unordered sets. Nevertheless, it describes the same hierarchy.

Clavii uses jBPT [65], an Open Source graph analysis framework, for the generation of a RPST. Therefore, the Activiti BpmnModel is converted into jBPTs representation of a BPMN 2.0-based process model (`Bpmn<BpmnControlFlow<FlowNode>`). For further runtime optimizations this conversion step can be bypassed by directly applying the RPST generation algorithm on a process model.

6.4.3 Process Model Graph Utilities

The *GraphUtils* library in Clavii offers often used graph algorithms (cf. Table 6.1). The development of own graph utilities was necessary, because Activiti does not offer any of these methods.

The library uses a RPST, it is divided into *GraphUtilsRPSTBase* and *GraphUtilsImpl*. *GraphUtilsRPSTBase* implemented methods to manage and query a RPST (e.g., calculate a RPST for a ClaviiBpmnModel, find RPST node types, convert a RPST node into a process model node set), while *GraphUtilsImpl* uses all these methods to execute higher-level algorithms (e.g., method *leastCommonSESE* calculates a minimal SESE block for a set of process nodes).

Method	Description
$getPredecessors(P, n)$	Returns a list of node IDs, which are precedent of process node n in process model P.
$getSuccessor(P, n)$	Returns a list of node IDs, which are successive of process node n in process model P.
$getCorrespondingGateway(P, n)$	Returns the node ID of the corresponding gateway for gateway n in process model P.
$getNodeDepth(P, n)$	Calculates the node depth of process node n in block-structured process model P. Start events and end events are on node depth 0.
$getTopologicalId(P, n)$	Returns the topological ID of a process node n in process model P, nested SESE blocks are sorted by branch ID
$getLeastCommonSESE(P, N, allowCommonSESE)$	Returns the least common SESE block of a set of process nodes N in process model P. If $allowCommonSESE$ is set to false, $getLeastCommonSESE$ returns `null`, if process nodes in N are not in the same minimal SESE block.
$isInSameBlock(P, n_1, n_2$	Checks, if two process nodes n_1, n_2 reside in the same minimal SESE block (i.e., the same branch)

Table 6.1: Clavii GraphUtils Methods Excerpt

6.4.4 Process Model Creation

A ClaviiBpmnModel can be created by invoking the method *ClaviiBpmnModelFactory.createClaviiBpmnModel(BpmnModel bpmnModel)*. This method creates a new ClaviiBpmnModel object and executes the following sub-methods: *correctInternalFlows*, *calculateRPST*, *checkBasicLayout*, *addBlockStructureExtensions*, and *calculateTopologicalMap*.

Method *correctInternalFlows* ensures a correct representation of an Activiti process model. Every process node contains maps of incoming and outgoing control flows, which may be wrong or incomplete due to a preceding change operation. Method *calculateRPST* calculates the RPST as descibed in [66]. Method *checkBasicLayout* checks the following conditions (with the help of the previously generated RPST):

- Process model contains exactly one start event
- Process model contains exactly one end event,
  as multi-terminal process models are not allowed in Clavii

- It exists an even quantity of gateways,
  because every gateway must have a corresponding gateway to be in conjunction with the block-structural constraints

- All control flows are completely connected (i.e., attributes SourceReference and TargetReference are set)

Otherwise, an exception will be thrown to indicate, that the process model to be converted does not comply with the specified Clavii layout. Method *addBlockStructureExtensions* creates ExtensionAttributes (cf. Section 5.2.3) containing information, like the ID of the corresponding gateway, and adds them to all gateways. Finally, method *calculateTopologicalMap* creates a topological map with all process node IDs as key and the corresponding topological ID as value.

6.4.5 Process Model Change Operations

Process model change operations in Clavii are divided into *atomic* and *compound* operations. While atomic operations only insert, change or delete a single element of a process model, compound operations are composed of two or more atomic operations. An example for a compound operation is $insertGatewayBlock(P, n_p, n_s, BlockType)$, which creates two new gateways with attribute $BlockType$ (i.e., `PARALLEL`, `EXCLUSIVE`, `LOOP`), a control flow between them and inserts all three process elements into process model P between preceding process node n_p and succeeding process node n_s.

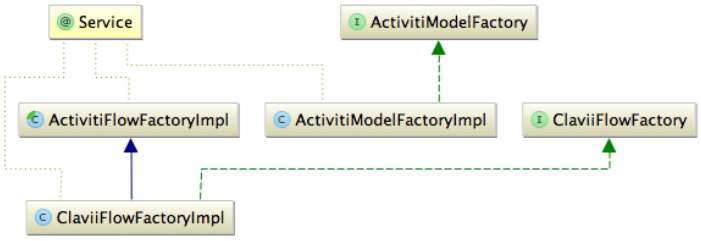

Figure 6.9: Clavii Process Model Factories UML Class Diagram

Atomic operations are located in class *ActivitiFlowFactory*, compound operations in class *ClaviiFlowFactory*. New process elements, like service tasks, may be created through class *ActivitiModelFactory* (cf. Figure 6.9).

Atomic Operations

Atomic operations insert, update or delete a single *FlowElement* in a *BpmnModel*. These are marked as *unchecked*, if the method does not contain any checks to ensure a correct application. Unchecked atomic operations directly alter a set of *FlowElements* in a *BpmnModel*, and therefore destroy structural consistencies (e.g., by adding a task without connecting it to other process nodes). Checked atomic operations also take care of correcting the control flow.

Insert operations are defined generic, such that *insertFlowNode* may insert every type of *FlowNode* (i.e., UserTasks, ServiceTasks, SequenceFlows) in either *sequence*, *parallel*, or in a new *branch*. This parameter is called **SequenceMode** in Clavii. Atomic operations are not visible within Clavii's API, as they do not implement correctness checking.

Compound Operations

Compound operations are higher-value operations invoking multiple atomic operations. They need strong transaction management. Table 6.4 gives a brief overview of all available compound operations. The latter are designed to be fail-safe and always create a correctly updated process model. Compound operations are visible through Clavii's API and may be used within a *ModelChangeDescription*.

Compound operation $deleteBlock$ can be executed with the ability to preserve containing process nodes and just remove all gateway nodes instead of all process nodes containing the block. Preservation is archived by serializing all branches considering each process node's topological ID.

Compound Operations are defined on a user's expectation. However, certain compound operations act differently. When applying compound operation $deleteBlock$ on a process model, a user may expect that this deletes a whole SESE block with its inherited process

nodes and a control flow between them. Clavii, in turn, only deletes the surrounding gateways and serializes all originally defined branches. This offers the ability to delete only surrounding process nodes, while inherited process nodes are preserved. If a user wants to delete a complete SESE block with all process nodes, he has to select all process nodes to be deleted. Internally, not method $deleteBlock$ is executed, but method $deleteProcessNode$, which again executes method $deleteProcessElement$ for every selected process element.

ModelChangeDescription

Changes on process models executed via the API are described by class *ModelChange-Descriptions*. A *ModelChangeDescription* consists of an **ID** referring the affected process model, a **ChangeOperation** object, a **NodeSet** object, a **ConfigurationMap** object and a **PropertyMap**. The *ChangeOperation* is a Java enumeration object, which defines an atomic or compound operation to be applied. $insertGatewayBlock$ is one example (cf. Tables 6.3 and 6.4).
The **NodeSet** object is represented as list of Strings, whereas all Strings are IDs for a specific *FlowNode* object in a process model. Class **ConfigurationMap** consists of *FlowElement*-Attributes, represented as *ElementAttribute* enumeration value (cf. Table 6.2). Each atomic or compound operation needs a different set of configuration values.
ModelChangeDescriptions are used, because they abstract updates of process models within Clavii. Therefore, every *ModelChangeDescription* may be converted to XML and used, regardless of which communication method is chosen between the Clavii Controller and a Clavii Client (e.g., client communication based on REST or Java Beans).

6.4.6 Process Model Update Procedure

A process model in Clavii can be updated by creating a *ModelChangeDescription* (cf. Section 6.4.5). A *ModelChangeFactory* exists to simplify creating the correct description, this factory also checks for parameter inconsistencies and a correct quantity of ElementAttributes (cf. Table 6.2). Afterwards the method `updateModel(BpmnModel, ModelChangeDescription)` at the *ProcessModelManager* is invoked (cf. Figure 6.10). The ProcessModelManager is defined to process and return `BpmnModels`, after invoking a method every model is converted into a *ClaviiBpmnModel* object (cf. Section 6.4.4).

Name	Description	Applicable Super Class
ID	Node ID	*BaseElement*
NAME	Item name visible for a user	*FlowElement*
DOCUMENTATION	Attribute representing a textual documentation of a element	*FlowElement*
CONDITION_EXPRESSION	Condition for a *SequenceFlow* after a *ConditionalGateway*. The branch will only be activated, if resolving the condition is `true`. Example for a condition in UEL: "$input == 1"	*SequenceFlow*
QUESTION	Question shown to a user when using the *manual* gateway mode	*Gateway*
ANSWER	Answer shown for a specific *SequenceFlow*	*Gateway*
DEFAULT_FLOW	Definition of the default *SequenceFlow*, referencing a sequenceFlowId	*Gateway*
TASK_TYPE	Defines a TaskType (`userTask`, `scriptTask`, `manualTask`, `scriptTask`). Switch for *insertFlowNode* method	*Activity*
SCRIPT	Script definition, which is executed when a *ScriptTask* is executed	*ScriptTask*
SCRIPT_FORMAT	Indicates the format of a stored Script, must be compatible with JSR-223 [32]	*ScriptTask*
RESULT_VARIABLE	Name of the variable in which a execution result will be stored	*ScriptTask*
AUTO_STORE_VARIABLE	Indicates, whether variable values will be stored automatically in the process model, currently not used by Clavii	*ScriptTask*
IMPLEMENTATION	Indicates a set of attributes containing a delegateExpression to invoke a Clavii plugin. Attributes must include *pluginName*, *pluginVersion* and *methodName*	*ServiceTask*
IMPLEMENTATION_TYPE	Defines, which invocation type is used for a service (*class*, *expression*, *delegateExpression*). Clavii usually invokes plugins with a *delegateExpression*	*ServiceTask*
RESULT_VARIABLE_NAME	Name of the variable in which the execution result will be stored	*ServiceTask*
ASYNCHRONOUS	Indicates, if a *ServiceTask* can be executed asynchronously. A succeeding Task will be executed, even if a ServiceTask did not finished, when `ASYNCHRONOUS` is *true*	*ServiceTask*
CANDIDATE_USERS	Set of users allowed to execute the Task	*UserTask*
CANDIDATE_GROUPS	Set of groups and roles allowed to execute the Task	*UserTask*

Table 6.2: Clavii ElementAttributes

Name	Description	Required Parameters and Element Attributes
$insertProcessNode(P, n, n_p, n_s, SequenceMode)$	Inserts process node n between process nodes n_p and n_s into process model P.	*ElementAttributes* according to the type of process node.
$insertGateway(P, BlockType)$	Inserts a new gateway into process model P. ID and name of the gateway are assigned automatically. Automatic naming is a legacy result, because Clavii's UI layouting algorithms depend on gateway names.	*BlockType*.{`PARALLEL, EXCLUSIVE, LOOP`}.
$insertControlFlow(P, e, name, n_{source}, n_{target})$	Inserts a new control flow edge e with optional name $name$ into process model P.	*ElementAttribute*.`NAME`
$insertEvent(P, name, EventType)$	Inserts a new event with event type $EventType$ and name $name$ into process model P. The ID of the event is assigned automatically.	*ElementAttribute*.`NAME` Parameter *EventType*.{`START, END, BOUNDARY, THROW`}.
$updateProcessElement(P, n, E)$	Applies a set of process element attributes E on process element n in process model P.	Allowed *ElementAttributes* for the class *FlowElement*.
$deleteProcessElement(P, n)$	Deletes process element n in process model P.	

Table 6.3: Clavii Atomic Operations

Every ModelChangeDescription contains a ChangeOperation Enum, which includes a abstract method `triggerUpdate(ModelChangeDescriptionDispatcher, ClaviiBpmnModel, ModelChangeDescription)`. This method is implemented by every Enum value and called by the `ProcessModelManager` with a new instance of a `ModelChangeDescriptionDispatcher` implementation (there are two different for ProcessModel and ProcessFilter operations). The `ModelChangeOperationDispatcher` reads all required `ElementAttributes` from the `ModelChangeDescription` and invokes the correct method in `ClaviiFlowFactory`.

6.5 PQL Proof-of-Concept Implementation

In order to set a focus the proof-of-concept implementation realizes a subset of the PQL functionality (cf. Section 6.5.2). Furthermore, instead of using the PQL meta-model, the PQL implementation works directly on Activiti's process model notation. The conversion of a PQL request into intermediate representation class PQLDescription (cf. Section 6.5.2) is independent from the Clavii BPM platform, while the business logic (with process model

Name	Description	Required Parameters and Element Attributes
$aggregateProcessNodes(P, N, name)$	Aggregates a set of process nodes N as virtual node with element name $name$ in process model P	*ElementAttribute*.NAME
$insertProcessNodeWith\text{-}Gateways(P, n_p, n_s, BlockType)$	Inserts a new process node with two surrounding gateways between process nodes n_p and n_s into process model P. Parameter $BlockType$ defines, whether the gateways to be inserted are either parallel, or exclusive gateways—or define a loop with two exclusive gateways.	Parameter *BlockType*.{PARALLEL, EXCLUSIVE, LOOP}.
$insertBranch(P, n_p, n_s, name)$	Inserts a new branch with name $name$ between two gateway nodes n_p and n_s into process model P.	*ElementAttribute*.NAME.
$insertGatewayBlock(P, n_p, n_s, BlockType)$	Inserts two *gateways* and a *control flow* in between into process model P. Process nodes n_p and n_s confine the insertion area of the gateway block.	Parameter *BlockType*.{PARALLEL, EXCLUSIVE, LOOP}.
$moveProcessNodeIntoNew\text{-}Branch\ (P, n_{move}, n_p, n_s)$	Moves a single process node n_{move} into a new branch between two gateway nodes in process model P. Therefore, attribute n_p defines the split gateway—the corresponding join gateway is calculated accordingly.	
$moveProcessNodeParallel\text{-}ToNode(P, n_{move}, n_p, n_s)$	Moves a single process node n_{move} parallel to a surrounding node n_s in process model P. Therefore, two new parallel gateways are inserted to surround the particular node.	
$moveProcessNode(P, n_{move}, n_p, n_s)$	Moves a process node n_{move} between process nodes n_p and n_s in process model P.	
$moveProcessNodes(P, N_{move}, n_p, n_s)$	Same behavior as $moveProcessNode$, but moves a complete SESE block described by a set of process nodes N_{move} between process nodes n_p and n_s.	
$toggleGateway(P, n_{toggle})$	Changes a gateway type of a gateway n_{toggle} from EXCLUSIVE to PARALLEL and vice versa. Control flow conditions will be removed, if the compound operation toggles the gateway type from EXCLUSIVE to PARALLEL. Missing Conditions (i.e., when toggling PARALLEL gateways) is retained by executing the gateway in MANUAL execution mode by default.	
$updateControlFlow\text{-}Conditions(P, e, condition)$	Applies control flow condition $condition$ on a process edge e in process model P.	*ElementAttribute*. CONDITION_EXPRESSION.
$deleteProcessNode(P, n)$	Deletes a process node n in a process model P.	
$deleteBlock(P, n_p, n_s, DeleteMode)$	Deletes a complete SESE block defined by preceding process node n_p and succeeding process node n_s in a process model P, if attribute $DeleteMode$ is set to value REMOVE. Otherwise, only two gateways (i.e., denoting entry and exit process node of the SESE block) are deleted.	Parameter *DeleteMode*. {INLINE, REMOVE}.
$deleteBranch(P, e)$	Deletes a single, empty branch e in process model P.	

Table 6.4: Clavii Compound Operations

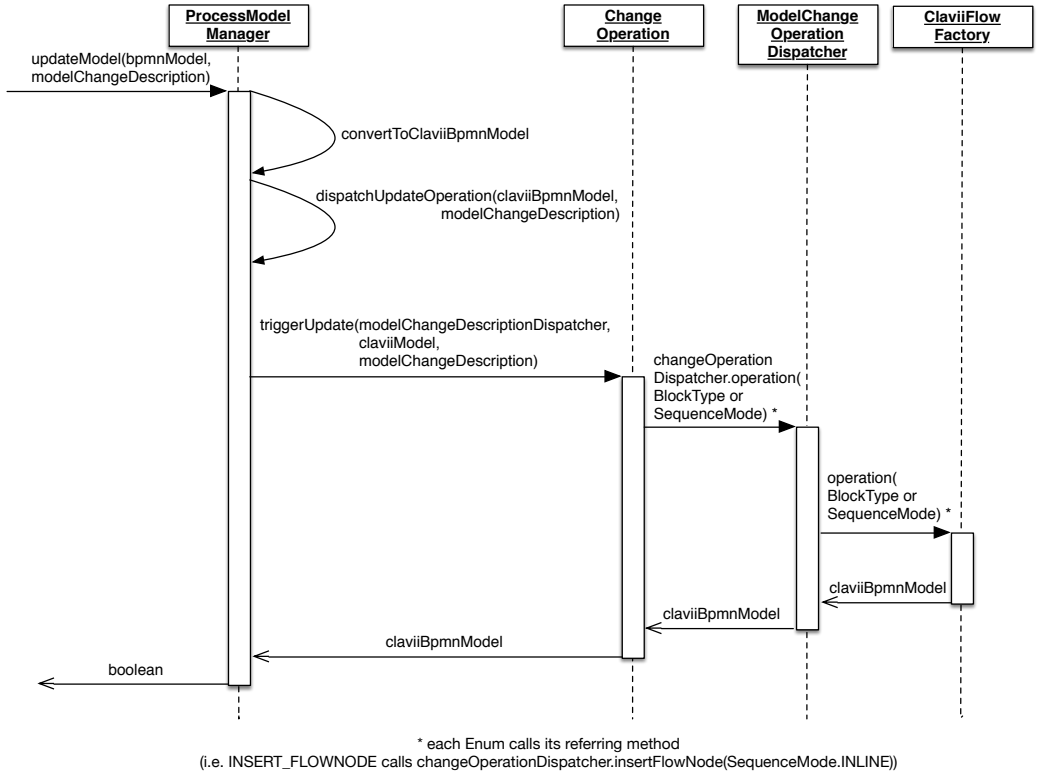

Figure 6.10: UML Sequence for Process Model Change Dispatching

change and view creation operations) is based on Clavii classes. Discovery step as described in Section 4.2.1 is implemented in a way, that a PQL request has either a defined reference on a process model identifier, or PQL requests are executed with reference on a Java process model object.

Section 6.5.1 gives a short introduction into the ANTLR parser generator, which is used to transform a PQL request into a PQLDescription, a machine-usable format. The format and all other elements are described in Section 6.5.2, while section 6.5.3 takes a look at the transformation procedure of a PQL request. Business logic applied in Clavii is topic of section 6.6.

6.5.1 Generating a Parser for Domain Specific Languages

Lexer and *parser* are necessary to convert a PQL request into a machine-readable format. A *lexer* or lexical analyzer is a software component, that creates *tokens* out of a sequence of characters (*String*) based on specified rules. This text must have a specific *syntax* in order to correctly create tokens. Tokens are grouped strings with a special meaning. A parser, in turn, converts these tokens to a semantic model (so-called *intermediate representation*).

The proof-of-concept implementation uses ANTLR (ANother Tool for Language Recognition), a *parser generator* [63]. The latter offers the ability to dynamically generate lexers and parsers based on a *grammar*, instead of developing them from scratch. ANTLR is written in Java and generates recursive descent parsers: parsing is executed from the root element of a parseable string to the leaves (tokens). Generally, an ANTLR grammar consists of four abstract computer language patterns: *sequence*, *choice*, *token dependence* and *nested phrases* [63]. A sequence (of characters) is a token (e.g., reserved identifier like GET, POST, PUT, or DELETE in the HTTP-Protocol [27]). Sequences are grouped by *rules*. `methodPost:'POST';` describes a *phrase*, consisting of a rule *methodPost* and an assigned token *POST*. The rule has to be executed, when the token occurs in the parsed String.

Furthermore, a rule may include *choices* between multiple, alternative phrases. By using phrase `methodPost:'POST'|'PUT';`, the rule *methodPost* has to be executed, when one out of the two tokens is present. Tokens may have dependent tokens. This occurs, for example, in case a grammar requires that both the opening bracket and closing bracket have to be present in a sequence. This dependency can be expressed by phrase `methodList:'('(method)+')';`, where *method* is another rule, that must occur between two tokens '(' and ')'. Phrase *'(method)+'* expresses, that rule *method* may occur exactly once or more. Finally, rules can refer to themselves. The latter is called a *nested phrase*. Phrase `expr:'a"('expr+')'|INT;` defines a nested phrase, that allows for recursive definitions. Sequence *a(5)* or even sequence *a(a(a(5)))* are valid expressions and match rule *expr*.

```
1 // Main Context
  pqlStatement
3     : (modelGetContext filterDeclarationModel? formatDeclaration? SEMICOLON)+ EOF
```

```
    ;
// 0 Context Declarations
modelGetContext
    :   GET MODEL sourceDeclaration
    ;
//    [...]
// 1 Source Declaration
sourceDeclaration
    :    Identifier
    ;
//    [...]
// 3 Filter Declaration
filterDeclarationModel
    :   FILTER (aggregationContext | reductionContext)+
    ;
reductionContext
    :   REDUCE LEFT_PAREN nodeAttribute RIGHT_PAREN exclusion?
    |   REDUCE LEFT_PAREN nodeAttribute (COMMA nodeAttribute)? RIGHT_PAREN exclusion?
    ;
//    [...]
// 4 Format Declaration
formatDeclaration
    :   FORMAT formatType
    ;
formatType
    : ( JSON
      | XML
      )
    ;
//    [...]
// X Reserved Keyword Tokens (Lexer Part)
Identifier
    :   Simple_Latin_Upper_Case_Letter
    // [...]
    ;
// 3.12 Operators
ASSIGN          : '=';
GT              : '>';
LT              : '<';
//    [...]
// a fragment is an abstract phrase, that must be defined in a phrase again
fragment Simple_Latin_Upper_Case_Letter
    :   'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' // [...]
    ;
```

Listing 6.3: PQL ANTLR Grammar Excerpt

With the help of a generated ANTLR parser based on a grammar, every parseable string is converted into a parse tree. Figure 6.11 shows such a parse tree for the grammar in

Listing 6.3. All uncapitalized nodes (except for leafs) of the parse tree represent *rules*. In contrast, capitalized nodes and leafs are *tokens*. This parse tree is interpreted by either a parse tree walker or parse tree visitor, ANTLR is able to generate both. Walkers and visitors are interfaces used to separate parser code from business logic. A walker fires events when a node in a parse tree is entered or left. These events can be caught to execute own methods, similar to a XML SAX parser [14].

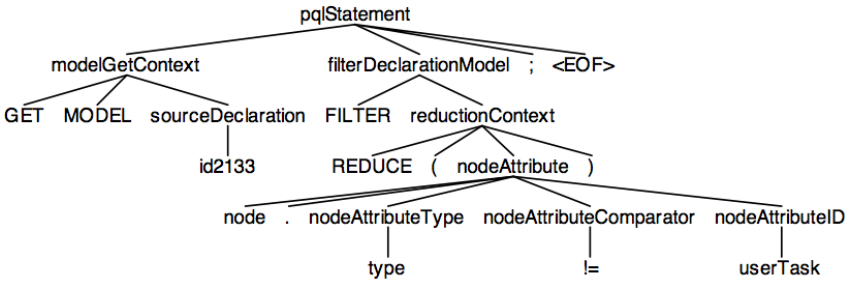

Figure 6.11: PQL ParseTree Example

In contrast, a visitor has the ability to control the parse tree walk. As a result, it is possible to call child nodes in a parent, similar to a XML DOM parser [60]. The latter emphasized as very useful for PQL (cf. Section 6.5.3).

6.5.2 PQL Request Representation

The PQL proof-of-concept consists of different components. It is divided into *contexts* to decide, which operations should be executed. Each context has a unique set of change operations described in the following.

Context

The PQL proof-of-concept earmarks more than the process modeling context. It can be used to manage run-time aspects as required by Requirement GRQ-1 (cf. Section 3.3). There are four different contexts: *repository*, *modeling*, *instance* and *monitoring*. Context *repository* is used to load and store process models based on a process model id. Context *modeling* offers methods to execute process model update operations implemented in

Clavii (cf. Section 6.4.5). The *instance* context allows to start new process instances while *monitoring* context is planed to deliver instance execution information, like the number of finished tasks or an execution trace log.

PQL Syntax

The proof-of-concept external DSL is defined by an ANTLR grammar, which describes lexer and parser definitions combined (cf. Section 6.5.1). A PQL request is a string, which has to be in conjunction with the PQL grammar. The PQL parser supports more than one PQL string per request. This feature is required to retrieve a set of process models by one PQL request. The PQL grammar is built hierarchically and starts with a root parser rule *pqlStatement*, which is built with the following schema: `<Context> <ChangeDescription> <FilterDescription> <FormatDescription>;`

The context consists of methods to retrieve, update, or delete a process model or alter a process instance. PQL request `GET MODEL id244`, for example, returns a process model with id=id244 from the process repository. PQL request `UPDATE MODEL id INSERTNODE(id,name,pred,succ)` inserts a new task with *id*, *name* between process nodes with id=pred and id=succ.

Class *FilterDescription* denotes reduction and aggregation operations based on process node attributes, to define on which process nodes these reductions should take place. An PQL example for the retrieval of a process model with applied process view creation operations is: `GET MODEL id FILTER REDUCE(id=3,id=4) AGGREGATE(user=${myself})`. Thereby, "${myself}" is a variable, that is replaced by the *id* of the executing user automatically. There are more organizational variables, like the organizational unit or organization of the user. Possible process node attributes are listed in Table 6.5.

Finally, class *FormatDescription* defines the return type of a process model, which may be either XML, or JSON (i.e., `GET MODEL id FORMAT JSON`). This feature may be used by web applications executed in a JavaScript engine. The latter is implemented in web browsers to provide a highly optimized environment for JSON format processing.

PQL Description

For easy processing of PQL requests within computer languages another representation of a PQL string is necessary. Class *PQLDescription* is an intermediate, object-oriented representation that represents exactly one PQL request.

Change operations are referred to process node IDs, whereas view creation operations are defined by process node attribute constraints.

Class *PQLDescription* consists of a model ID, the original PQL string, the context (as enumeration data type), the desired output format (enumeration data type, and values XML or JSON) and two lists of *PQLFilterDescription* and *PQLChangeDescription* objects.

A *PQLFilterDescription* consists of a `FilterType` (i.e., values `REDUCE`, `AGGREGATE`), a set of filter attributes and a list of node IDs to be excluded from abstraction. Excluded nodes are, for example, nodes, that are changed by a preceding change operation. Otherwise, newly inserted nodes may be reduced and, thus, not shown to a user. If defined, filter attributes match these newly inserted nodes.

A filter attribute contains of a `FilterAttributeType` attribute, a `FilterAttribute-ComparatorType` attribute and a value to be compared against. Both are enumeration types and describe, on which node attribute (i.e., attribute type in Table 6.5) and with which comparator function (i.e, attribute comparator: EQUALS, NEQUALS, LIKE) a node attribute should be checked against the value. `FilterAttribute(NODE_USER, LIKE, "Peter")` expresses for example, that an abstraction will be applied on every node, whose assigned user attribute contains value "Peter". The Clavii implementation, therefore, looks up all registered users in service OrgModelManager, that contain value "Peter" and also checks against their IDs. This is necessary, since Activiti stores only user and group IDs as assignment attribute values in a node.

6.5.3 Parsing and Conversion Procedure

In order to convert a PQL request into a PQLDescription object, four different classes are involved: *PQLExecutor*, *PQLVisitor*, *ParseTree* and an instantiated *PQLDescription*

Variable	Node Attribute
NODE_ID	ID of a process node.
NODE_NAME	Visible name of a process node.
NODE_TYPE	Node type of a process node(i.e., user task, service task).
NODE_PRED	Predecessor process node ID, filter actions are applied on the succeeding process node.
NODE_SUCC	Successor process node ID, filter actions are applied on the preceding process node.
NODE_USER	Assigned process participant of a process node, implementation supports a process participant's ID as well as its common name as value.
NODE_GROUP	Assigned group of a process node, implementation supports group ID and group name as value.

Table 6.5: PQL FilterAttributeType

(cf. Figure 6.12). Class *PQLExecutor* is responsible for calling methods in the ANTLR framework to convert a PQL request into an intermediate representation, called parse tree. A parse tree is a undirected graph tree, in which every node represents a previously parsed rule (as defined in the PQL grammar). Each detected parse tree node calls method `visit(ParseTree)`, that executes methods, depending from the parse tree node type, implemented in class PQLVisitor to modify the PQLDescription object. Figure 6.11 shows an example parse tree for the PQL request `GET MODEL id2133 FILTER REDUCE(node.type != userTask).`

The Clavii BPM platform makes use of class PQLExecutor and converts the declarative PQLDescription class into process model change operations (cf. Section 6.6).

6.6 Process View Implementation

Process views (cf. Section 3) are named *Process Filters* in Clavii and are executable with the *ProcessFilterManager*. A Process Filter can be either a PQL request String or a predefined request, called *FilterDefinition*. This *FilterDefinition* consists of a PQL request, a custom name and information used by the UI (i.e., icons). *FilterDefinitions* are stored by Clavii's persistence manager (cf. Section 6.3).

6.6.1 Creating a Process View

Creating a process view the *ProcessFilterManager* requires a reference on a process model. This may either be commited by a PQL request containing such reference ("GET MODEL

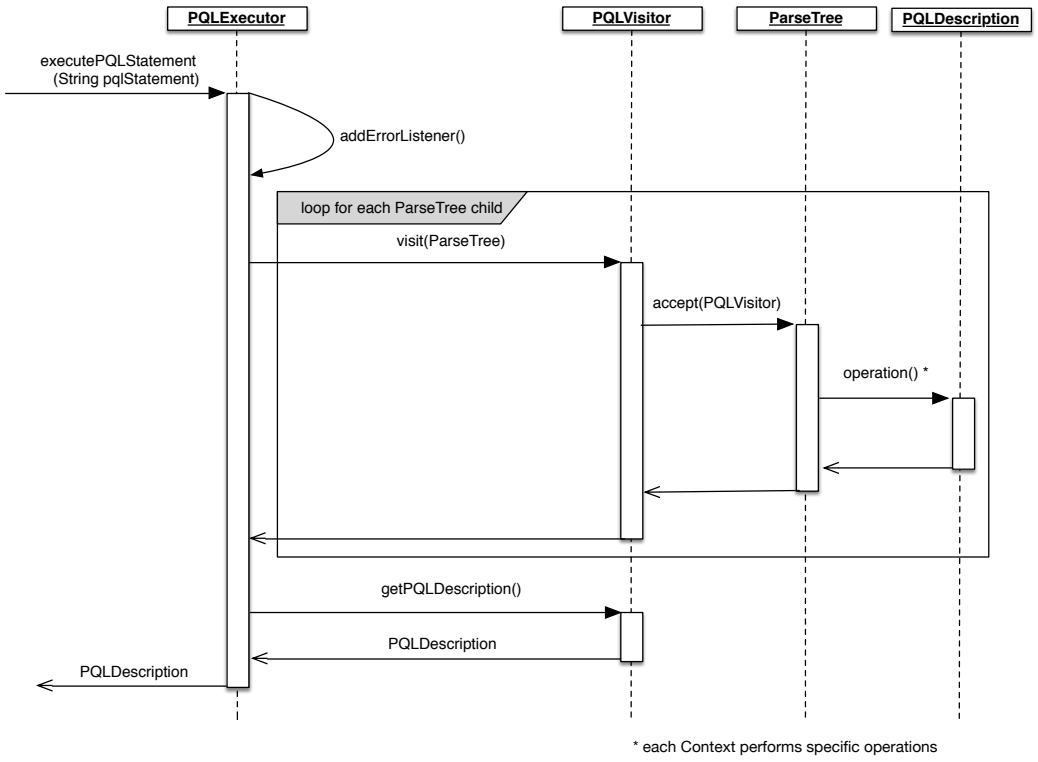

Figure 6.12: UML Sequence Diagram for converting a PQLDescription Object

<id>", cf. Section 6.5.2) or a method call with a reference on the respective process model object.

Figure 6.13 shows the conversion procedure by an UML sequence diagram. A generation is started with the method call `executeFilter(BpmnModel, FilterDefinition, Agent)` on the *ProcessFilterManager*. The PQL request (embedded in a *FilterDefinition*) is first converted into a *PQLDescription* object. Subsequently, filter variables are set on the *PQLDescription*, which are necessary to consider a users context for filter application (which user is logged in, what organization does he belong to). Table 6.6 shows available variables and their description.

Afterwards, the *PQLDescription* object is converted into a *ViewDefinition* object containing a set of *ChangeOperation* objects. Every reduction or aggregation request is converted

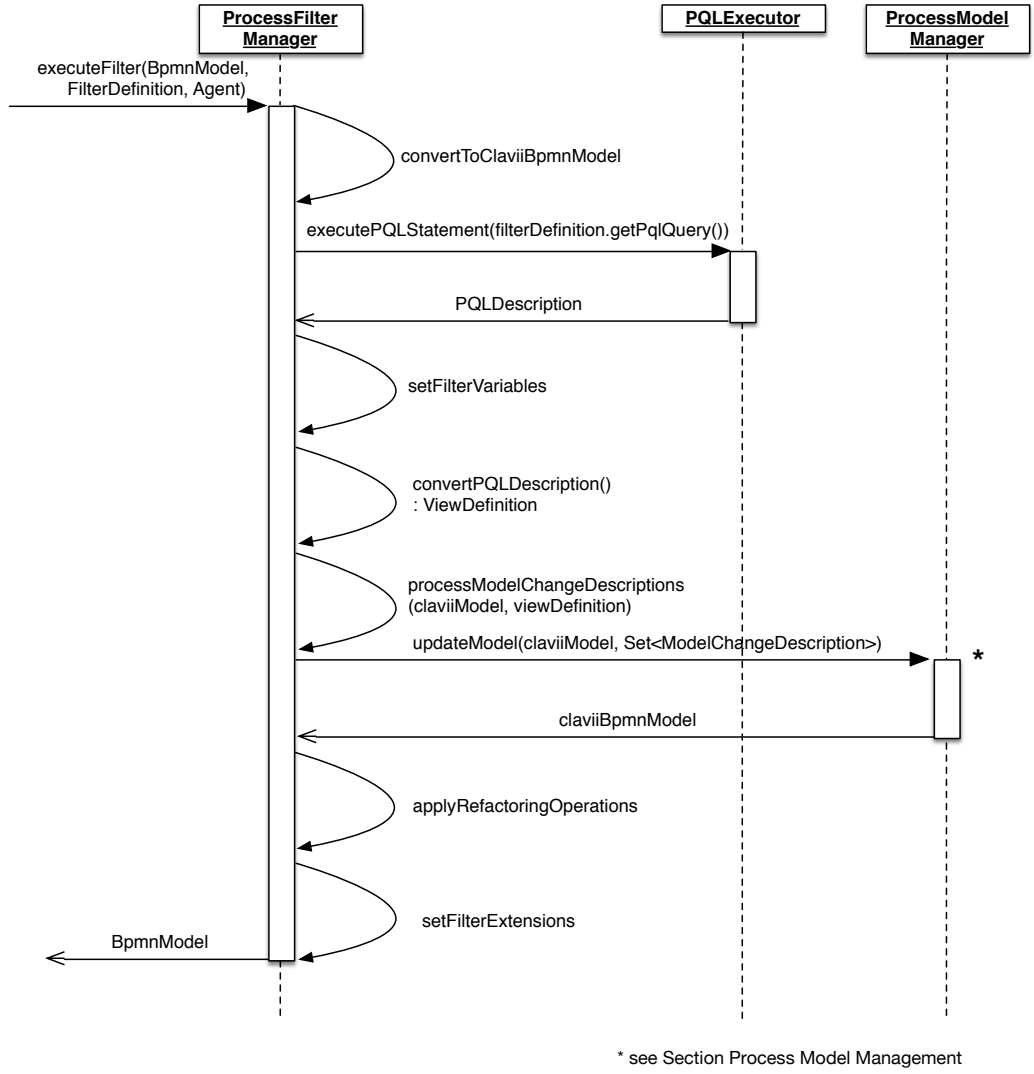

Figure 6.13: UML Sequence of a Process Filter Execution

into one *ChangeOperation* object. The required node sets are generated by logically concatenating *PQLFilterAttributes* and parsing the process model for process nodes these attributes fit on. Node identifiers, names and types are linked by a logical OR-every process node must comprise at least one attribute to be added to the node set.

Name	Description
${model}	The ${model} filter variable is replaced by the *ClaviiBpmnModel.Id* the filter is executed on.
${myself}	The ${myself} filter variable is replaced by the id of the executing process participant.
${mygroup}	The ${mygroup} filter variable is replaced by the group id of the executing process participant.
${myorg}	The ${myorg} filter variable is replaced by the organization id of the executing process participant.

Table 6.6: Clavii Process Filter Variables

Subsequently, the *ViewDefinition* is applied to the process model with the *ProcessModelManager* (cf. Section 6.4.6), the require call contains the *ClaviiBpmnModel* and the set of *ProcessModelChangeDescriptions* (cf. Section 6.4.5).

Refactoring operations are applied on the generated process view to reduce empty blocks or multiple branches. Finally, *FilterExtensions* are added to the process view. These extensions enable other components to comprehend executed view operations (cf. Listing 6.2).

Figure 6.14 and Figure 6.15 show applied FilterDefinitions on the process model in Figure 6.7. The FilterDefinition in Figure 6.14 reduces all process tasks, except technical, like service or script tasks, while the filter in Figure 6.15 preserves user tasks. As we can see, the original process model is much larger and difficult to understand.

The next section illustrates updates on a process view, for which *FilterExtensions* are assessed.

6.6.2 Updates on Process Views

Updates applied on process views in Clavii are based on the algorithms described in Section 3.1.3 and run as characterized in Section 3.1.3.

First, method `updateOnFilter(ClaviiBpmnModel, FilterDefinition, Agent, ModelChangeDescription)` in class *ProcessFilterManager* is called. *ModelChangeDescription* contains update *ChangeOperations* to be executed on the CPM *ClaviiBpmnModel*. These process model change operations contain node sets, which could include

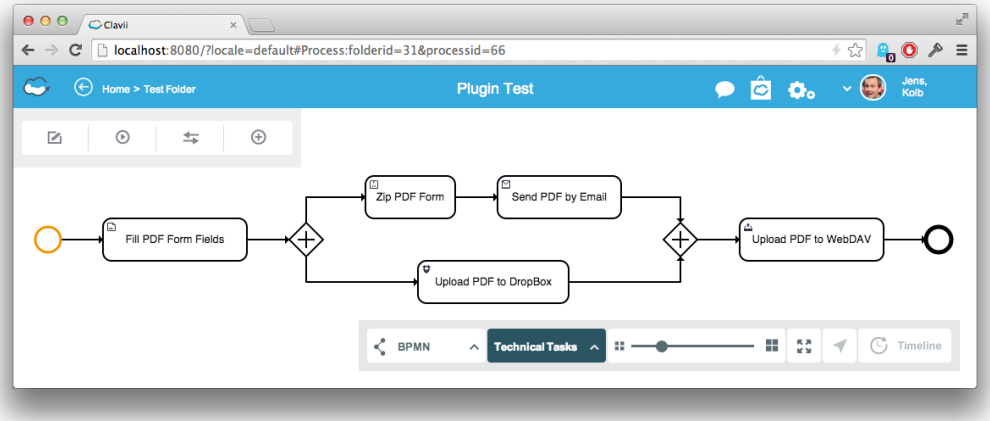

Figure 6.14: Clavii Technical Task Process View

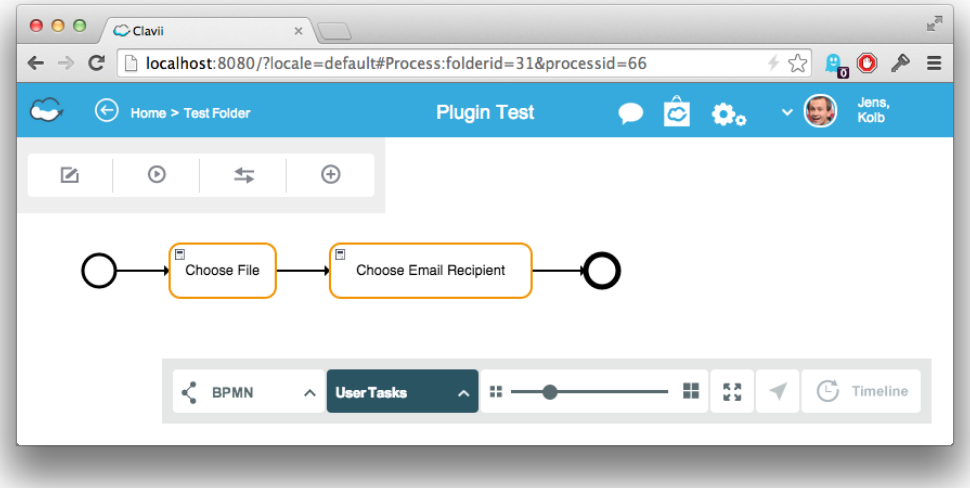

Figure 6.15: Clavii User Task Process View

virtual nodes or insert operations not regarding surrounding reduced nodes. Thus, every *ModelChangeDescription* is corrected by calculating and comparing node sets out of the *FilterDefinition*. Hence, virtual nodes may be resolved to their comprising process nodes.

Arising ambiguities are solved by parameter *ParameterMode* (cf. Section 3.1.1). Parameter *ParameterMode* is set to default values in the proof-of-concept implementation, but may be customized by users. Parameter *ParameterMode* is a Java interface. The latter is implemented by other parameters (e.g., parameter *InsertSerialMode*, cf. Figure 6.16).

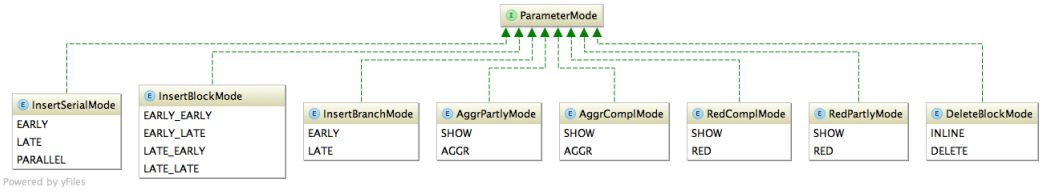

Figure 6.16: ParameterMode Type Hierarchy

Furthermore, after correction of all *ModelChangeDescriptions*, view update operations are executed on the CPM (i.e., implemented as *ClaviiBpmnModel*) like common process model change operations (cf. Section 6.4.6). Hence, the *CPM* has been changed, the *FilterDefinition* has to be recalculated to create an updated process view. To prevent the view creation from reducing newly inserted nodes and, thus, not showing changes of a process view, every updated node ID is added to an exclusion node set, which is present in every *PQLDescription*. Finally, the updated view is created according to section 6.6.1 and returned to the method caller.

6.7 Summary

In this section the Clavii BPM platform proof-of-concept implementation was introduced. It is built with a MVC-like architecture running as a JEE Application. Clavii offers various functionalities, for example identity management, process instance monitoring, validation management for process models, or a plugin architecture to be able to use additional service tasks. Each is described by a manager interface.

Section 6.4 delineated, how process models are represented, stored and changed. Process model changes in Clavii are built hierarchically to avoid code redundancies and are based on the GraphUtils library. The latter utilizes a RPST to determine block-structures, which

are required by Clavii's process model. The PQL proof-of-concept implementation enables Clavii to define process views declaratively. Pre-defined PQL requests can be applied on arbitrary process models and, thus, are process model independent. Furthermore, updates on process views are supported by Clavii, as well as refactoring operations removing not necessary process nodes after creating a process view.

The next Section takes a look at work related to process views and PQL.

7

Related Work

Proviado is an approach to create process views enabling personalized views on process models and process instances [9, 69]. It offers methods for *structural* and *graphical adaptions* of process models. *Structural adaptations* in Proviado involve process view creation algorithms to manipulate representations of control flow, process element attributes, or process data elements. Figure 7.1 shows an example of view operation `AggrShiftOut` *Aggregate(B,C,H,K)*.

Graphical adaptions comprise template mechanisms to configure the graphical representation of process models.

The proView approach is based on emerged results of Proviado and offers additional support to update process views and related CPMs directly eventuating in control-flow and view creation limitations [40, 41, 44]. In particular, process models have to be block-structured, arising ambiguities during process view modifications require additional parameters.

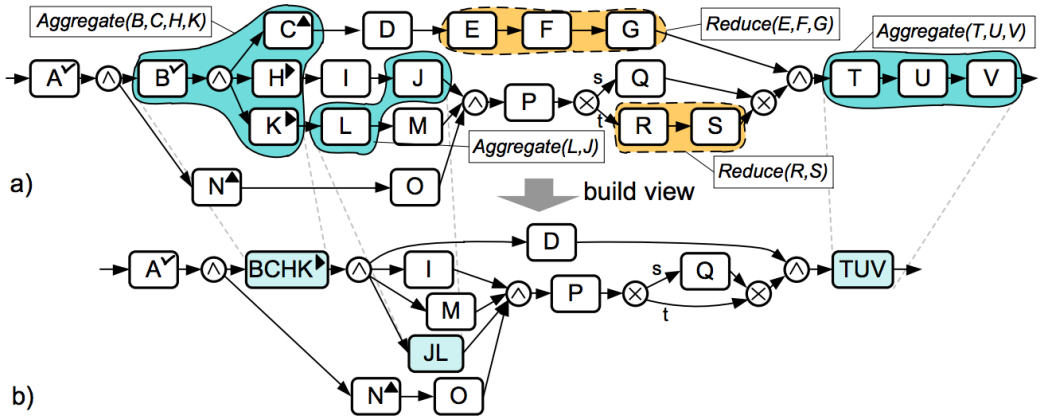

Figure 7.1: Proviado Process View Example (Source: [69])

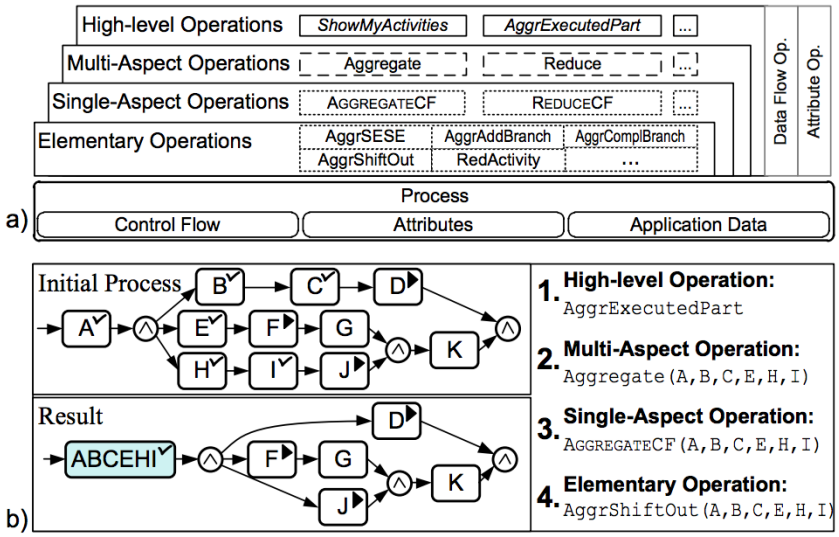

Figure 7.2: ProView Multi-Layer View Operations (Source: [69])

The proposed concepts for PQL-based process views and its implementation in the Clavii BPM platform are based on both approaches. Dynamic process views result from high-level view operations described in Proviado and proView (cf. Figure 7.2). However, PQL enables the ability to define process views in a declarative way, rather than strictly relating to a specific process model (cf. Section 3.1.3). During view creation, mappings between virtual

nodes (to be created) and process nodes can be determined and adjusted dynamically by evaluating PQL queries. Process views defined with PQL are not referring to a specific CPM. Instead, abstract properties, like task types or relations between nodes, are used. As a result, PQL queries allow to apply view operations on multiple, dynamically discovered models. Explicit view definitions are also possible with PQL, as sets of node IDs can be defined.

Sakr et al. proposed a framework for querying process models [73]. BPMN-Q uses parts of the BPMN 2.0 meta-model, its queries are built as fragments with specialized objects. Its language meta-model offers different elements divided into meta-classes (cf. Figure 7.3). The *Connectivity* meta-class, for example, contains *paths* and *sequence flows*. *Sequence flows* connect two adjacent process nodes, while a *path* only denotes a *path* between two nodes, i.e., there might be other nodes in between. The *Activity* meta-class describes generic process activities. *Abstract events* differentiate between start, intermediate and end events and are summarized in the *Event* meta-class. Finally, gateway nodes in another meta-class distinguish between split and join parallel, conditional and OR gateways.

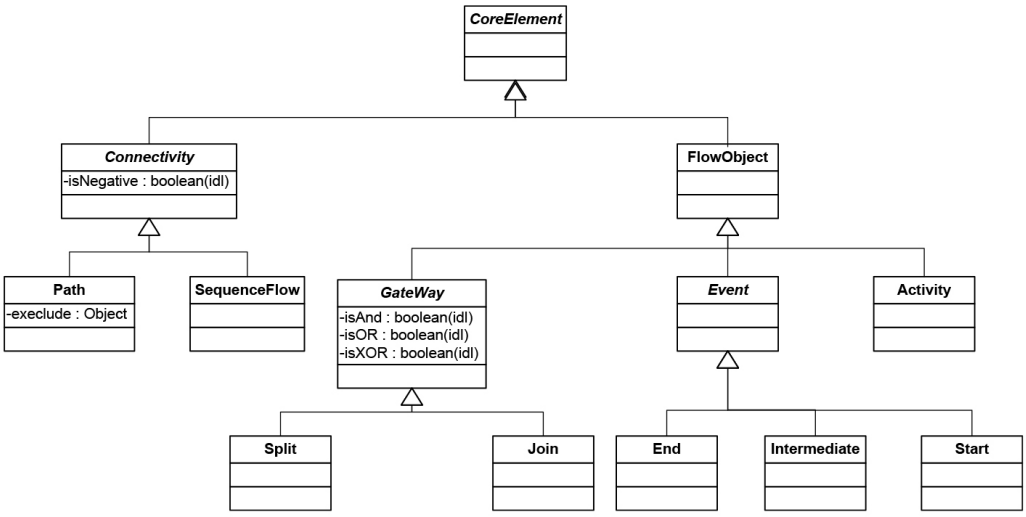

Figure 7.3: UML Class Diagram for BPMN-Q Meta-Model (Source: [5])

Furthermore, BPMN-Q offers a visual interface to define queries, which consist of the above described elements. In addition, such a query may contain other elements to describe variables to be discovered.

Matching a query against process models is done by path discovery. Therefore, every searchable process model has to be preprocessed and stored in a RDBMS with a fixed-mapping storage scheme. Graphical queries are converted into semantical expanded queries using a *Semantic Query Expander*. The *SQL-based Query Processor* then executes SQL scripts based on the semantical expanded queries to discover the RDBMS for matching fragments.

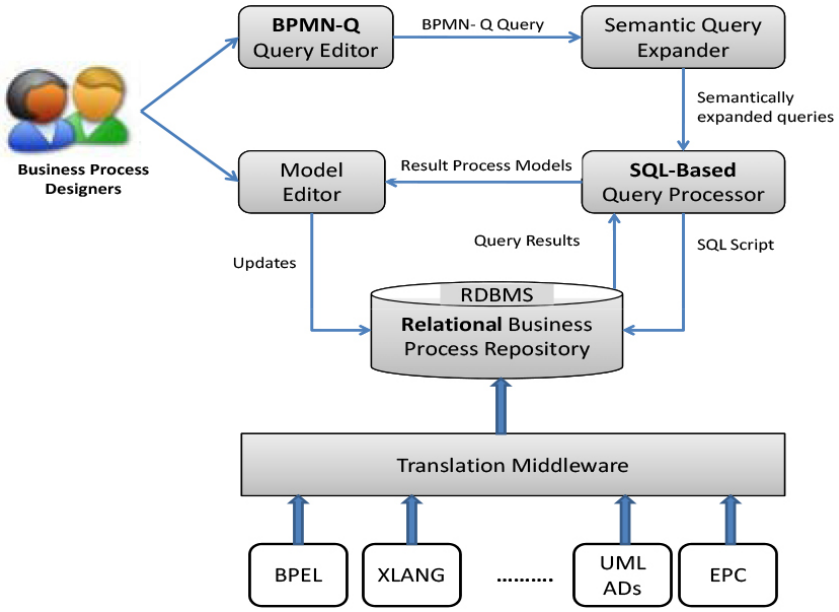

Figure 7.4: BPMN-Q Framework Architecture (Source: [73])

BPMN-Q's meta-model only supports a behavioral perspective, other process elements like data elements are not discoverable. Process node attributes are not considerable as well. BPMN-Q targets only queries to search for-updates on process models and abstractions are not supported. A discovery is based on variables, which have to be defined in a query. Thus, process similarity search approaches (i.e., by edit-distance) useful for process variants are not supported.

BP-QL is a GUI-based language allowing for querying process models [7, 8]. Its graphical notation is based on *state charts*. Hence, a query is defined by state chart patterns. Additional state chart elements are used to define data flows between process activities. BP-QL uses pattern matching on node attributes and control flows to search for process models matching a query fragment. Control flows can also be designed to allow place holders. Therefore, edges with one arrow may not have additional nodes inbetween a search fragment, while two arrows describe, that a correct search result may have additional nodes in between two nodes of the search fragment. Dashed edges and nodes of a query fragment denote a negation as constraint, a negated path may not exist in a process model. Figure 7.5 illustrates a BP-QL query requireing the search component to be reached without prior login.

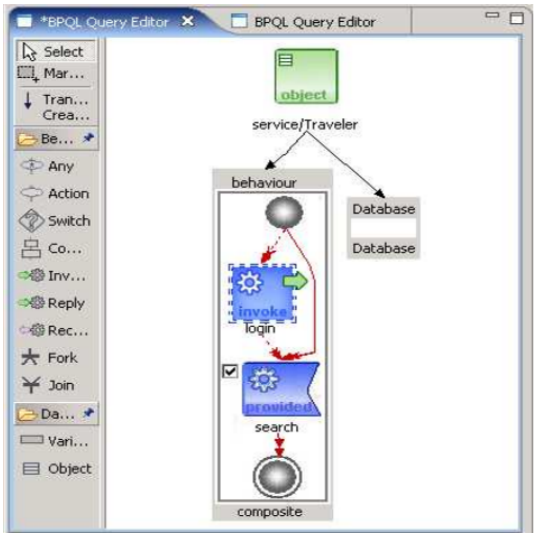

Figure 7.5: BP-QL Query Example (Source: [8])

The proof-of-concept implementation is implemented in Active XML intended to query BPEL process models and returning data from embedded web service calls. Therefore, Active XML offers to define data explicitly, as usual for XML, and *intentionally* to be able to obtain data dynamically [2]. Intentional data definitions are useful for BPEL, because embedded web service calls may return a variety of not explicitly defined data types.

When searching for process models, first BP-QL queries are translated into XQuery queries, which are then executed on BPEL definitions.

BP-QL also treats some kind of process abstraction, it offers "different levels of granularity" either showing "fine-grained" or "coarse-grained" process models for higher level abstraction. "Fine-grained" granularity zooms into process models and shows additional information, "coarse-grained" granularity depicts process nodes expressing external web service calls as "black boxes" [8].

8

Conclusion

This thesis presents PQL, an approach for a process query language to simplify business process modeling. It features a processing pipeline offering business process discovery based on exact matching or similarity matching, process modeling capabilities to update process models and process abstractions. The PQL meta-model is able to map arbitrary graph-based process modeling notations. Thus, common tasks, like process model changes or process abstractions do not have to be implemented discretely for every notation, but may be executed by functionalities supplied by PQL.

Next steps for PQL are the development of a technical standard and a reference implementation to show interoperability features between different PAIS implementations. Additionally, open issues about process model similarity discovery exist, like how to transform process models for efficient process discovery.

The Clavii BPM platform is a web-based and integrated PAIS. It simplifies the BPM lifecycle and enables rapid development of executable process models. The Activiti BPM platform proves to be very flexible - during the development of Clavii various components of Activiti are extended. This involves process gateway logic and adding ad-hoc changes for manual decisioning of branches during execution, just to mention two extensions.

Beside Clavii a component is created, where common graph algorithms are implemented based on the RPST. It allows to calculate standard tasks used for process modeling and process abstraction, which Activiti and other PAIS do not support out of the box. A simplified creation and management approach of process views is implemented based on this graph utility component and the proof-of-concept implementation of PQL. Process views described by PQL are more flexible and easier to implement, while enabling high-level process view operations.

Clavii's approach shows a new way to simplify the workload in the context of BPM. It may be extended to support distributed process engine instances [52] and execution flexibility for process tasks, as well as dynamic load balancing required to build a flexible, scalable, and simple to use BPM cloud platform [23].

A

Appendix

This appendix contains source codes written in Java.

A.1 Activiti Code Examples

Listing A.1 shows a simple example, where the Activiti engine is configured and started (Lines 1-3), required services instanciated (Lines 5-8), a process model imported (Lines 10-13) and instantiated (Lines 15f).

```
1 public void startEngine() {
  ProcessEngine processEngine = ProcessEngineConfiguration
3       .createStandaloneInMemProcessEngineConfiguration()
        .buildProcessEngine();
5 // instantiate services
  RuntimeService runtimeService =
7       processEngine.getRuntimeService();
```

```
RepositoryService repositoryService =
      processEngine.getRepositoryService();
// import process model from XML file
repositoryService.createDeployment()
      .addClasspathResource(
          "newProcessDef.bpmn20.xml")
      .deploy();
// start process instance for previously imported process model
ProcessInstance processInstance =
      runtimeService.startProcessInstanceByKey("newProcessDefName");
System.out.println("id " + processInstance.getId() + " " +
      processInstance.getProcessDefinitionId());
}
```

Listing A.1: Activiti Code Sample Engine Initialization

Listing A.2 shows a code example on how to create a new Task, save it to the TaskService, create a new User with the IdentityService, define users allowed to execute the newly created task and finally execute it.

```
@Autowired
TaskService taskService; // TaskService is Autowired by Spring
@Autowired
IdentityService identityService;

public void createAndExecuteTask() {
    // create and save generic Task
    Task task = taskService.newTask();
    task.setName("New Task");
    taskService.saveTask(task);

    // create, save and assign User
    User user = identityService.newUser("FrancisUnderwood");
    // it is also possible to create a group
    // Group newGroup = identityService.newGroup("marketing");

    identityService.saveUser(user);
    taskService.addCandidateUser(task.getId(), "FrancisUnderwood");

    // claim and complete Task as User
    taskService.claim(task.getId(), "FrancisUnderwood");
    taskService.complete(task.getId());
}
```

Listing A.2: Activiti Code Sample Creation and Execution of a Task

In order to execute a service task, Activiti offers four possibilities:

- POJO[1] Java Service Task Class
    - with or without field extensions
    - with method or value expressions
- Java Delegate (resolves to a registered Java Bean)

The first possibility uses a simple Java class, which has to implement the `JavaDelegate` interface provided by Activiti. This interface defines a method called `void execute (DelegateExecution execution)`, in which the business logic may be implemented.

[1] Plain Old Java Object

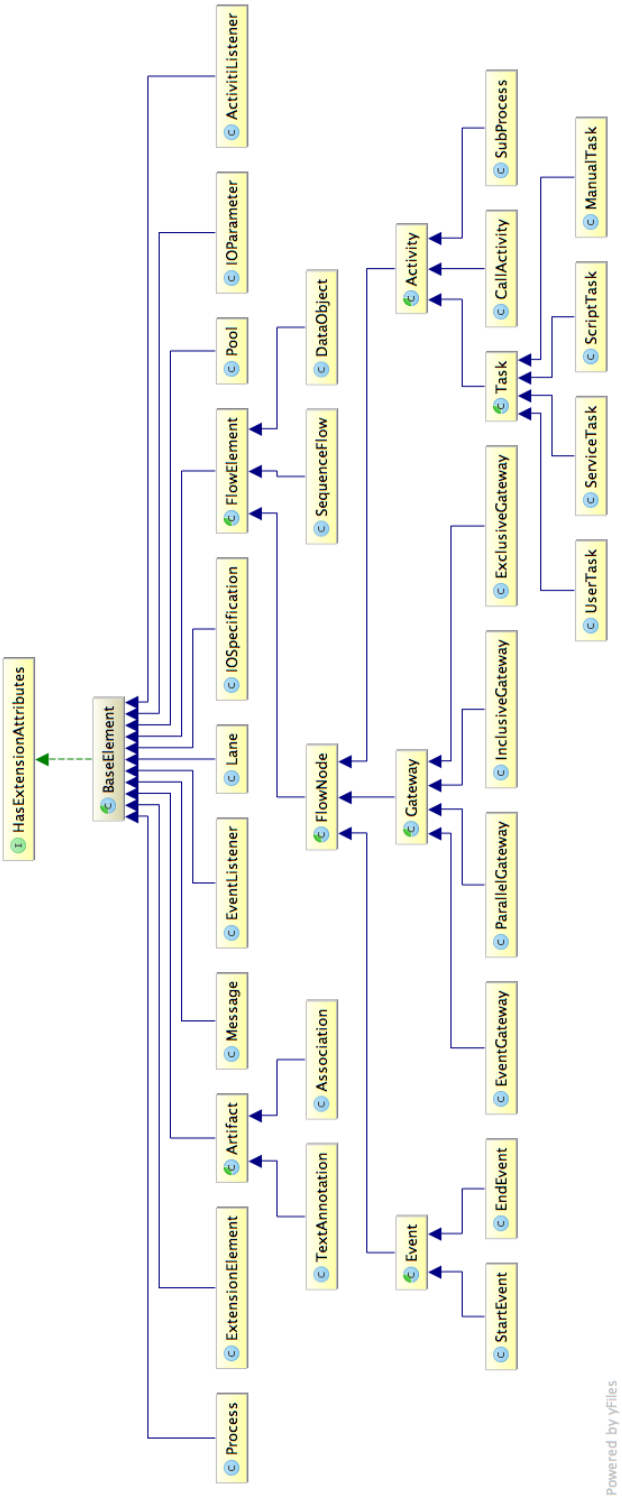

Figure A.1: Activiti BaseElement UML Class Diagram Excerpt

Glossary

API Application Programming Interface. Specifies interfaces between software components.

BPM Business process management. Systematical approach to capture, execute, document, measure, monitore and control automated and non-automated processes to reach goals defined by a business strategy.

BPMN 2.0 Business Process Model and Notation. Graphical representation to specify business processes by process models.

Business Intelligence Theories and technologies to convert raw data into meaningful information. Business Intelligence is used to handle huge amounts of process data to identify and optimize business opportunities.

Business Process A set of tasks, that have to be done in a predefined order to achieve a particular target.

Business Rule Description for operations, definitions, and constraints that apply to an organization.

CPM Central process model. Process model used as base for process views.

Creation Set Specifies the schema and appearance of a process view.

DSL Domain-specific language. A computer programming language with focus on a specific problem domain.

Framework A universal, reusable software component to develop software applications providing generic functionality easily expanded by application-specific logic.

Java Bean Serializable and reusable Java class with public methods, that encapsulates many Java objects. Used in a Java EE environment.

Java Class Extensible template, that defines initial variable states and programming behaviors.

JEE Java Platform, Enterprise Edition. A software programming framework for developing and running distributed enterprise applications.

JUEL Java implementation of UEL.

Open Source Product or system, whose ideas and design are freely available to the public.

PAIS Process-aware information system.

Process Designer Component of a PAIS to analyze and model a business process.

Process Fragment Sub-graph of a process model.

Process Model Schematic description of a business process.

Process Node Type of process element included in a process model. A node can either represent a task, gateway, event, or entity, like a process participant or document.

Process View Simplified process model abstracted by different process perspectives.

REST REpresentational State Transfer. Architectural style consisting of architectural constraints applied to components, connectors and data elements within a distributed hypermedia system.

Sequence Flow Directed edges connecting process nodes to express an execution order.

SESE Single-Entry, Single-Exit. A fragment of a process model, that has exactly one incoming and one outgoing edge.

Syntax Grammatical rules and structural patterns for an ordered use of words and symbols to express software methods and data.

UEL Unified Expression Language. A special purpose programming language, that simplifies access to data objects by an easy syntax.

View Creation Operation Operation, which transforms a process model into a process view.

View Update Operation Operation to update a process view and its associated CPM.

XML eXtensible Markup Language. Markup language, that defines a set of rules to encode documents in a format, that is human- and machine-readable.

List of Figures

List of Tables

Listings

Bibliography

[1] H2 Database Engine. http://www.h2database.com/, last visited: 08-26-2014, 2014.

[2] S. Abitrboul, O. Benjellourn, I. Manolescu, T. Milo, and R. Weber. Active XML: Peer-to-peer Data and Web Services Integration. In *Proceedings of 28th International Conference on Very Large Data Bases*, pages 1087–1090. VLDB, 2002.

[3] O. Alliance. *OSGI Service Platform, Release 3*. IOS Press, Inc., 2003.

[4] K. Andrews. Design and Development of a Run-time Object Design and Instantiation Framework for BPM Systems. Master's thesis, Ulm University, 2014.

[5] A. Awad. BPMN-Q: A Language to Query Business Processes. In *EMISA*, volume 119, pages 115–128, 2007.

[6] F. Bayer and H. Kühn. *Prozessmanagement für Experten: Impulse für aktuelle und wiederkehrende Themen*. Springer, 2013.

[7] C. Beeri, A. Eyal, S. Kamenkovich, and T. Milo. Querying Business Processes. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 343–354. (VLDB) Endowment, 2006.

[8] C. Beeri, A. Eyal, S. Kamenkovich, and T. Milo. Querying Business Processes with BP-QL. *Information Systems*, 33(6):477–507, 2008.

[9] R. Bobrik. *Konfigurierbare Visualisierung komplexer Prozessmodelle*. PhD thesis, University of Ulm, 2008.

[10] R. Bobrik, T. Bauer, and M. Reichert. Proviado – Personalized and Configurable Visualizations of Business Processes. *E-Commerce and Web Technologies*, 4082:61–71, 2006.

[11] R. Bobrik, M. Reichert, and T. Bauer. Parameterizable Views for Process Visualization. Technical report, University of Twente, 2007.

[12] R. Bobrik, M. Reichert, and T. Bauer. View-Based Process Visualization. In *5th Int'l Conf. on Business Process Management (BPM'07)*, LNCS, pages 88–95. Springer, 2007.

[13] Brooks, R.J. and Tobias, A.M. Choosing the Best Model: Level of Detail, Complexity, and Model Performance. *Mathematical and Computer Modelling*, 24(4):1–14, 1996.

[14] D. Brownell. *SAX2*. O'Reilly Media, 2002.

[15] S. Büringer. Development of a Cloud Platform for Business Process Administration, Modeling and Execution. Master's thesis, Ulm University, 2014.

[16] E. H. Chi. A Taxonomy of Visualization Techniques using the Data State Reference Model. In *Information Visualization, 2000. InfoVis 2000. IEEE Symposium on*, volume 94301, pages 69–75, 2000.

[17] M. Cumberlidge. *Business Process Management with Jboss Jbpm*. From Technologies to Solutions. Packt Publishing, 2007.

[18] P. Dadam and M. Reichert. The ADEPT Project: a Decade of Research and Development for Robust and Flexible Process Support. *Computer Science-Research and Development*, 23(2):81–97, 2009.

[19] T. H. Davenport and J. E. Short. The new Industrial Engineering: Information Technology and Business Process Redesign. *Sloan Management Review*, 31(4):11–27, 1990.

[20] T. De Bruin and M. Rosemann. Towards a Business Process Management Maturity Model. In *ECIS 2005 Proceedings of the Thirteenth European Conference on*

Information Systems, pages 1–12, Germany, Regensburg, 2005. London School of Economics.

[21] L. DeMichiel and B. Shannon. JSR 342: Java^{TM} Platform, Enterprise Edition 7 (Java EE 7) Specification, 2013.

[22] R. Dijkman, M. Dumas, and L. García-Bañuelos. Graph Matching Algorithms for Business Process Model Similarity Search. In *Business Process Management*, volume 5701 of *LNCS*, pages 48–63. Springer, 2009.

[23] E. F. Duipmans, L. F. Pires, and L. O. B. da Silva Santos. Towards a BPM Cloud Architecture with Data and Activity Distribution. *2012 IEEE 16th International Enterprise Distributed Object Computing Conference Workshops*, pages 165–171, 2012.

[24] H. E. Eriksson and M. Penker. *Business Modeling with UML: Business Patterns at Work*. OMG. Wiley, 1998.

[25] J. Fawcett, D. Ayers, and L. Quin. *Beginning XML, 5th Edition*. ITPro collection. Wiley, 2012.

[26] P. Feldbacher, P. Suppan, C. Schweiger, and R. Singer. Business Process Management: A Survey among Small and Medium Sized Enterprises. In W. Schmidt, editor, *S-BPM ONE - Learning by Doing - Doing by Learning*, volume 213 of *Communications in Computer and Information Science*, pages 296–312. Springer, 2011.

[27] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616-HTTP/1.1, the Hypertext Transfer Protocol, 1999.

[28] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[29] J. Freund and K. Götzer. *Vom Geschäftsprozess zum Workflow: Ein Leitfaden für die Praxis*. Hanser, 2008.

[30] J. Freund and B. Rücker. *Praxishandbuch BPMN 2.0*. Carl Hanser Verlag GmbH & Company KG, 2012.

[31] M. C. Golumbic. Algorithmic Graph Theory and Perfect Graphs. *Academic, New York*, 1980.

[32] M. Grogan. JSR-223 Scripting for the Java TM Platform, 2006.

[33] A. Gupta. *Java EE 7 Essentials*. O'Reilly Media, 2013.

[34] J. N. Gupta, S. K. Sharma, M. A. Rashid, and I. Global. *Handbook of Research on Enterprise Systems*. Information Science Reference, 2009.

[35] G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2004.

[36] Information Technology – Database Languages – SQL – Part 11: Information and Definition Schemas (SQL/Schemata), 2011.

[37] R. Johnson, D. Pearson, and K. Pingali. The Program Structure Tree: Computing Control Regions in Linear Time. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 171–185. ACM, 1994.

[38] D. Karagiannis and P. Höfferer. Metamodels in Action: An Overview. *ICSOFT*, pages 11–14, 2006.

[39] U. Kargl. *Change Management im Business Process Management: BPM initiierte Veränderungsprozesse*. Diplomica Verlag, 2013.

[40] J. Kolb, K. Kammerer, and M. Reichert. Updatable Process Views for Adapting Large Process Models: The proView Demonstrator. In *Demo Track of the 10th Int'l Conf on Business Process Management (BPM'12)*, number 940 in CEUR Workshop Proceedings, pages 6–11, 2012.

[41] J. Kolb, K. Kammerer, and M. Reichert. Updatable Process Views for User-centered Adaption of Large Process Models. In *10th Int'l Conference on Service Oriented Computing (ICSOC'12)*, number 7636 in LNCS, pages 484–498. Springer, 2012.

[42] J. Kolb, H. Leopold, J. Mendling, and M. Reichert. Creating and Updating Personalized and Verbalized Business Process Descriptions. In *6th IFIP WG 8.1 Working Conference on the Practice of Enterprise Modeling (PoEM'13)*, number 165 in LNBIB, pages 191–205. Springer, November 2013.

[43] J. Kolb and M. Reichert. A Flexible Approach for Abstracting and Personalizing Large Business Process Models. *ACM SIGAPP Applied Computing Review*, 13(1):6–17, March 2013.

[44] J. Kolb and M. Reichert. Data Flow Abstractions and Adaptations through Updatable Process Views. In *Proceedings of 28th Annual ACM Symposium on Applied Computing (SAC 2013)*, pages 1447–1453, 2013.

[45] J. Kolb, M. Reichert, and B. Weber. Using Concurrent Task Trees for Stakeholder-centered Modeling and Visualization of Business Processes. In *S-BPM ONE 2012*, number 284 in CCIS, pages 237–251. Springer, 2012.

[46] M. Konda. *Just Spring*. O'Reilly Media, 2011.

[47] M. Konda. *Just Hibernate*. O'Reilly Media, 2014.

[48] U. Kreher. *Concepts, Architecture and Implementation of an Adaptive Process Management System*. PhD thesis, University of Ulm, 2014.

[49] A. Lanz, J. Kolb, and M. Reichert. Enabling Personalized Process Schedules with Time-aware Process Views. In *CAiSE 2013 Workshops, 2nd Int'l Workshop on Human-Centric Information Systems (HCIS 2013)*, number 148 in Lecture Notes in Business Information Processing (LNBIP), pages 205–216. Springer, 2013.

[50] C. Li, M. Reichert, and A. Wombacher. On Measuring Process Model Similarity Based on High-Level Change Operations. In *Conceptual Modeling-ER 2008*, pages 248–264. Springer, 2008.

[51] O. L. Madsen. *Block Structure and Object Oriented Languages*, volume 21. ACM, 1986.

[52] J. Mangler and S. Rinderle-Ma. Rule-Based Synchronization of Process Activities. *2011 IEEE 13th Conference on Commerce and Enterprise Computing*, pages 121–128, 2011.

[53] J. Marinacci. *Building Mobile Applications with Java: Using the Google Web Toolkit and PhoneGap*. O'Reilly Media, 2012.

[54] A. Marzal and E. Vidal. Computation of Normalized Edit Distance and Applications. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 15(9):926–932, 1993.

[55] J. Mendling, G. Neumann, and M. Nüttgens. A Comparison of XML Interchange Formats for Business Process Modelling. *Workflow Handbook*, pages 185–198, 2005.

[56] J. Mendling, H. A. Reijers, and W. M. van der Aalst. Seven Process Modeling Guidelines (7PMG). *Information and Software Technology*, 52(2):127–136, 2010.

[57] B. Meyer. Conception, Design, and Evaluation of a Graphical User Interface for a Cloud Platform for Business Process Management. Master's thesis, Ulm University, 2014.

[58] M. Minor, A. Tartakovski, and R. Bergmann. Representation and Structure-Based Similarity Assessment for Agile Workflows. In *Case-Based Reasoning Research and Development*, pages 224–238. Springer, 2007.

[59] B. P. Model. Notation (BPMN) Version 2.0. *OMG Specification, Object Management Group*, 2011.

[60] G. Nicol, L. Wood, M. Champion, and S. Byrne. Document Object Model (DOM) Level 3 Core Specification. 2001.

[61] C. Nock. *Data Access Patterns: Database Interactions in Object-Oriented Applications*. Addison-Wesley Boston, 2004.

[62] A. Osterwalder and Y. Pigneur. *Business Model Generation: A Handbook for Visionaries, Game Changers, and Challengers*. Wiley, 2013.

[63] T. Parr. *The Definitive ANTLR 4 Reference*. 2013.

[64] J. L. Peterson. Petri Nets. *ACM Computing Surveys (CSUR)*, 9(3):223–252, 1977.

[65] A. Polyvyanyy. jbpt - A Compendium of Process Technologies. https://code.google.com/p/jbpt/, last visited: 08-26-2014. Online; accessed 20-Jul-2014.

[66] A. Polyvyanyy, J. Vanhatalo, and H. Völzer. Simplified Computation and Generalization of the Refined Process Structure Tree. In *Web Services and Formal Methods*, volume 6551 of *LNCS*, pages 25–41. Springer, 2011.

[67] T. Rademakers. *Activiti in Action: Executable Business Processes in BPMN 2.0*. Manning Pubs Co Series. Manning Publications Company, 2012.

[68] M. Reichert and P. Dadam. Enabling Adaptive Process-Aware Information Systems with ADEPT2. In *Handbook of Research on Business Process Modeling*, pages 173–203. Hershey, New York, 2009.

[69] M. Reichert, J. Kolb, R. Bobrik, D. Ag, and T. Bauer. Enabling Personalized Visualization of Large Business Processes through Parameterizable Views. In *27th ACM Symposium On Applied Computing (SAC'12)*, 2012.

[70] M. Reichert and B. Weber. *Enabling Flexibility in Process-Aware Information Systems: Challenges, Methods, Technologies*. Springer, 2012.

[71] A. Rubinger and B. Burke. *Enterprise JavaBeans 3.1*. O'Reilly Media, 2010.

[72] J. Russell and R. Cohn. *Graphviz*. Book on Demand Ltd., 2012.

[73] S. Sakr and A. Awad. A Framework for Querying Graph-Based Business Process Models. In *Proceedings of the 19th International Conference on World Wide Web*, pages 1297–1300. ACM, 2010.

[74] A.-W. Scheer and M. Nüttgens. *ARIS Architecture and Reference Models for Business Process Management*. Springer, 2000.

[75] K. Shahzad, M. Elias, and P. Johannesson. Towards Cross Language Process Model Reuse – A Language Independent Representation of Process Models. *The Practice of Enterprise Modeling*, 39:176–190, 2009.

[76] D. Spinellis. Notable Design Patterns for Domain-Specific Languages. *Journal of Systems and Software*, 56(1):91–99, 2001.

[77] R. E. Tarjan and J. Valdes. Prime Subprogram Parsing of a Program. In *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 95–105. ACM, 1980.

[78] W. M. P. van der Aalst. Formalization and Verification of Event-Driven Process Chains. *Information and Software technology*, 41(10):639–650, 1999.

[79] W. M. P. van Der Aalst, A. H. M. Ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[80] B. Weber, M. Reichert, J. Mendling, and H. Reijers. Refactoring Large Process Model Repositories. *Computers in Industry*, 62(5):467–486, 2011.

[81] B. Weber, M. Reichert, and S. Rinderle-Ma. Change Patterns and Change Support Features–Enhancing Flexibility in Process-Aware Information Systems. *Data and Knowledge Engineering*, 66(3):438–466, 2008.

[82] M. Weske. *Business Process Management: Concepts, Languages, Architectures*, volume 14. Springer, 2012.

[83] L. Yujian and L. Bo. A Normalized Levenshtein Distance Metric. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 29(6):1091–1095, 2007.

[84] M. Zur Muehlen and J. Recker. How Much Language Is Enough? Theoretical and Practical Use of the Business Process Modeling Notation. In *Advanced Information Systems Engineering*, pages 465–479. Springer, 2008.

Name: Klaus Kammerer Matrikelnummer: 650769

Erklärung

Ich erkläre, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den .....

Klaus Kammerer