ulm university universität
uulm

Universität Ulm | 89069 Ulm | Germany

Fakultät für Ingenieurwissenschaften und Informatik
Institut für DBIS

The WfQL: A Proposal for a Standard WfMS Interface

Diplomarbeit an der Universität Ulm

Vorgelegt von:
Mark Oliver Schmitt
mark.schmitt@uni-ulm.de

Gutachter:
Prof. Dr. Reichert
Prof. Dr. Dadam

Betreuer:
David Knuplesch

2011

Fassung February 20, 2011

Contents

1 Introduction

Today, division of work has become common in the economic world. It is obvious, that an engineer, who developed a car, will not build it and that a surgeon will not nurse a patient before and after an operation.

This is the very essence of todays, in history unprecedented productivity per worker.

The focus of workers on specific activities requires them to implement a workflow, in which the individual worker executes activities they are specialized on. These activities are often based on the work of other workers. In the past, the work did not change that much over time and the model, that describes the individual workers position and work within a workflow, could be memorized.

Today, virtually every company is competing on the marked, which leads to optimization of workflows, and thus requires the individual worker to adapt quickly to new workflow models. With the computerization being available, many companies use digital documents instead of physically existing ones – which makes it possible for computers to communicate most of the data that is exchanged between the individual workers of a workflow. These aspects lead to the development of **Workflow Management Systems** (**WfMS**), which are computerized aids that help companies execute their workflows.

These systems are capable of reminding the individual workers about pending activities and allows them to focus on their work, as the flow of work is getting managed by WfMS. To illustrate our point, we are formulating the following example:

Fig. 1.1 shows a simple order to delivery process that consists of three steps:

1. A telephone operator in customer service recording orders from customers
2. A book-keeper who makes sure, the invoice is executed
3. Shipping of product

The figure also shows the flow of data between the **stakeholders**:

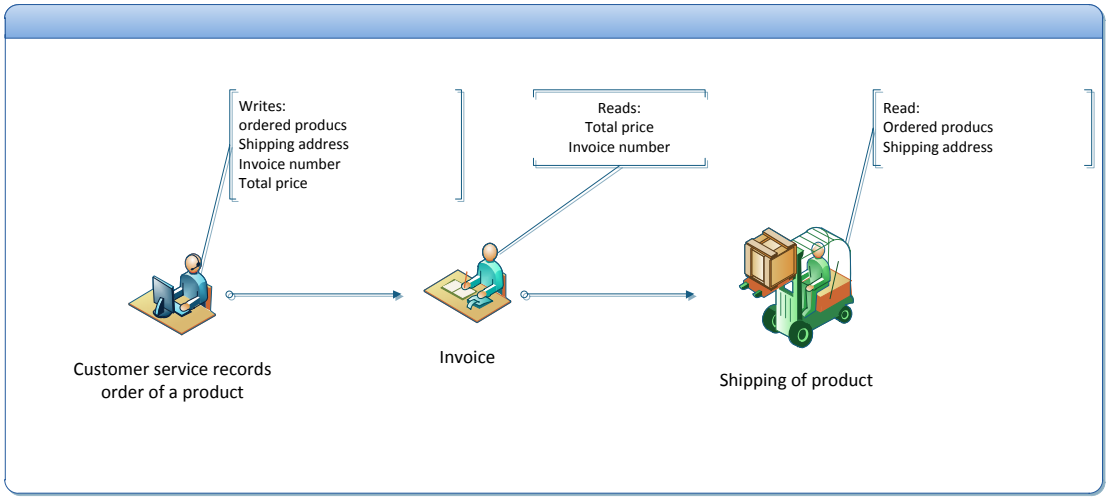


Figure 1.1: Example workflow with dataflow

- The telephone operator stakeholder notes the customers shipping address, the ordered products, calculates a total price and give the customer an invoice number for future reference.
- The book-keeping stakeholder gets informed about the total price and the invoice number
- The shipping stakeholder gets informed about the ordered products and the shipping address.

If a WfMS was used for the presented process, the telephone operator could enter the customer data, ordered products, shipping address and the customer billing number into the WfMS. The WfMS could store this information in the workflow and inform book-keeping and shipping with the billing number, ordered products and shipping address respectively.

Businesses execute many processes like the exampled one. All processes have in common, that they are repetitively executed tasks, that depend on input and generate output. Most processes have implicit rules that grew over time and are often not formalized prior to the use of a WfMS. They exist in the minds of the people working on them. In such an environment, when no formal step by step protocol is kept, the actual state – the current activity and the following activity – become diffuse. Information flows between the **stakeholders** using any means – voice, written notes both on paper or via electronic means.

When the rules that govern processes get formalized, there are many possibilities to do so: Diagrams such as Figure 1.1, full text depictions and step-by-step instructions are efforts to build models after which a process should be executed.

WfMS require formal models. Therefore, we first take a look at how business processes get executed when they are formalized: Each process gets executed by rules that define the order of activities that need to get executed. This formalization of a process defines the **process model**, the actual execution of processes according to a model is called a **process instance** of this model. Process models are given a name, the combination of a process model and a name forms the **process template**. New process instances are created based on process templates.

Similar to the real world, the instance's process model may vary at times from the template.

The description of activities in workflows without computerized aids can be implicit, without noting every single input necessary to full-fill the activity. However, with computerized systems, the input of activities has to be described exactly. Especially, when direct communication between the stakeholders is impossible.

Analogue to process templates, activities can be predefined as well. **Activity templates** define classes of activities that have common input and common output. A common activity therefore can be described just one time and then used in all models. Obviously, an activity requires specific input depending on the process instance it gets executed in, and return its output to this instance. The advantages of activity templates is enormous: An activity can be executed ad hoc, without knowing anything about the process it is part of, and by different stakeholders.

Stakeholders executing activity instances require an interface to get the activity's input and return the activity's output to the WfMS. This aid is usually supplied in the form of **activity programs**. They are either part of a WfMS or interface with one.

The rules defining the order of activities and decisions, whether certain activities get executed is called the **control flow**. In addition, the input and output of activities is handed through the process's execution by means of the so called **data flow**. The latter defines, which output becomes the input of a later activity or control flow decision.

The **stakeholders** we discussed in the initial example are executing activities. Other classes of stakeholders exist as well: stakeholders that create and change process models or analyze past and currently running processes. It is expected, that not all stakeholders

share one computerized system, but that they are using computers that can communicate with the WfMS via a computer network. Stakeholders use programs on their computers that execute their tasks, such as modeling processes or executing activities. These programs need to communicate with the WfMS via the network. Currently, no standard exists that governs this communication in all aspects. Therefore, the providers of WfMS develop their own communication protocols and languages, that are not compatible to competing WfMS.

The Migration Problematic

The available standardized protocols for WfMS specific information are not supporting all aspects of workflow specific communication. We therefore assume, that in the following scenario, a proprietary WfMS system is deployed.

We also assume, that there are users of such a system and it is used to implement many aspects a company's daily work: The system governs the employees' activities and models repetitively executed processes, like the shipping process shown in Figure 1.1. The employees' use computers with programs, that in turn use the WfMS. At some point in time, problems surface with the WfMS. A competing provider of WfMS offers a solution, and the company would like to migrate.

However, the new system is incompatible, because the the two WfMS use different protocols for communication. When switching to another protocol, the employees programs have to be adapted to the new protocol and process models have to be migrated, possibly manually by re-entering it with a new modeling program.

The cause of this problem is, that the two WfMS use different protocols to communicate. If they had used a common protocol, the company could have migrated to a new WfMS without the need to migrate the clients and could have automated the migration of data held in the WfMS.

The Binary Problematic

When developing programs that make use of a WfMS, a means of interfacing with the WfMS is required. Providers of WfMS thus supply customers with programs or libraries that supply this interface. They have to be deployed on the client computers, so they can be used by the client programs to interface with the WfMS.

However, the deployment of binary code is problematic in many ways: For one, the inner workings of the binary interface component is hidden from its users, which prohibits them from fixing bugs. Second, the binary code has to be compatible to the client computer. Only the target platforms that the provider decided to supply binary interfaces for are supported. Third, when the protocol changes, for example due to a new WfMS server version, all target platforms need to get updated.

Proposal for a solution

In order to solve the migration and the binary problematic, we propose the development of a communication language for WfMS.

To do so, we discuss requirements to a WfMS interface and to a underlying WfMS in chapter 2, name design decisions and assumptions we made on the underlying WfMS in chapter 3, present the concrete realization of a model of a WfMS implementing the design decisions in chapter 4 and present the language specifications in chapter 5. In chapter 6 we present a prototypical implementation of the language specified in the form of a server and a several client implementations.Chapter 7 discusses further possible enhancements to the language and discusses aspects, that require further research.

We name this language the **Workflow Query Language** (WfQL), due to its comparable usage scenario to the *Structured Query Language* for Database Management Systems.

2 Requirements

In this chapter, the requirements towards both interface, the WfQL, and the underlying WfMS are discussed. We formulate them from the stakeholder's perspective. The requirements towards the WfMS are in later chapters used to formulate a model, which WfMS have to comply to when implementing a WfQL compatible interface.

Not all requirements are towards the WfMS. In part, they are direct requirements towards the WfQL specifications. We differentiate between general requirements to both components, and specific requirements by naming the target component in the requirement definition if only one of them is concerned.

In chapter 3, these requirements are used to search for existing standards and specifications that could possibly satisfy them.

2.1 Requirements along the process life-cycle

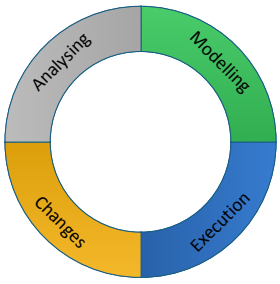


Figure 2.1: Lifecycle

The stakeholder's interactions with the WfMS are presented with the help of the process life-cycle. Stakeholders develop a new process template and design new activity templates or use existing ones in the **modeling stage**. It is followed by the **execution stage**, that is, the repetitive materialization of process templates in the form of process instances, which in turn causes activities to get executed by activity stakeholders. During the **change stage**, modifications to process models and activity templates are carried out. The **analysis stage** describes how analyst stakeholders examine currently executing and terminated processes and activities.

During the course of presenting this four staged life cycle, we note requirements to a WfMS and to the WfQL.

2.1.1 Modeling stage

During the modeling stage, processes are modeled with the following entities: The process model of process templates with data flow and control flow, activity templates and the organization structure.

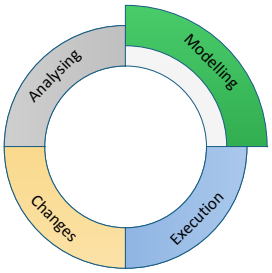


Figure 2.2: Modeling

Thus, we can state the first requirements:

- **Req 1**: Creation and persistence of process models of process templates and activity templates

The control flow should be flexible, so activities can be omitted and executed parallelly and repetitively:

- **Req 2**: The control flow of process models allows omitting activities and repetitive and parallel execution of activities

Activities require input and generate output. As such, the corresponding activity templates have to declare which input and output to expect:

- **Req 3**: Activity templates are modeled with input and output variables

Furthermore, only sound process models shall be persisted in the WfMS:

- **Req 4**: Ensure soundness

WfMS are used in companies that have organization structures and as such, we require the WfMS to reflect that structure when assigning activities to workers and as basis for permission management:

- **Req 5**: An organization model is used as basis for staff assignment and permission management

2.1.2 Execution stage

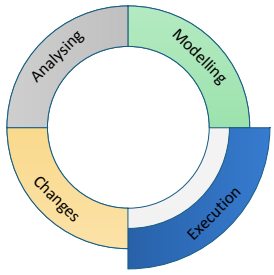


Figure 2.3: Execution

During the execution stage, the WfMS waits for stakeholders to initiate process instances based on process templates. This procedure is called **process instantiation** and results in a process instance.

- **Req 6**: Instantiation of new process instances based on process templates

The WfMS should then "run" the instance and execute it until it arrives in a termination state. During the execution, activity templates are referenced from the process model and supplied with input, which creates *activity instances*. They require execution by *activity stakeholders* who require the activity's input to execute them. Additionally, the WfMS has to ensure, that no two stakeholders execute the same activity instance. We name the procedure of accepting an activity instance with its input as **check-out** and the return of the output as **check-in**.

- **Req 7**: WfMS: the correct and conflict free execution of process instances is ensured

Process instances eventually terminate. Like activities, they may have an output that a stakeholder may be interested in. This can either be accomplished, by gathering return parameters directly by querying the WfMS for the results of a terminated process instance, or by instructing it to inform the stakeholder about finished processes:

- **Req 8**: The termination state of terminated process instances can be accessed
- **Req 9**: Stakeholders can register themselves to get informed about the termination state of process instances

2.1.3 Change stage

Processes and activities can change over time – new best practices, guidelines or laws may require immediate implementation. Therefore, it is necessary to support changes to the process model of process instances, process templates and to activity templates.

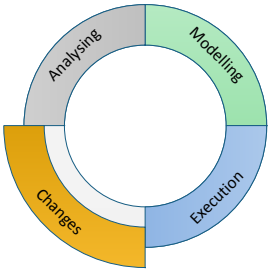


Figure 2.4: Changes

While changes on process templates get executed, a new process instance may get created based on this template. While changing a process template, it may either not get instantiated at all, or the WfMS has to isolate the changes.

The same issues surface with process instances. We need to address the following two problems:

1. Depending on the progress of the process instance's execution, possibly not all modification can be applied: already executed parts of graphs cannot be altered
2. Related changes need to be applied as one. If not, the execution could proceed between two modifications and prohibit applying the second change.

A very similar problem is faced with process templates. Multiple modifications that a *change stakeholder* sees as group have to be applied as one. If not, a process instance could get instantiated in-between alterations and execute a template that was never intended for execution. In the works we reference, these changes to process templates and process instances are often subsumed with the term **process evolution**.

When changing process instances, it is possible, that the execution continues while changes get applied. Thus, optional operations should be possible that allow for individual changes to be omitted if not applicable. Additionally, not only one instance may need alteration but multiple ones, applied at the same time.

When process templates are altered, it may become clear, that the process template should be split into two distinct process templates. Thus, we require a method to derive a copy from an original template to a new one, and migrating a part of the instances based on the original one to the new one.

In summary, we obtained the following requirements:

- **Req 10**: Activity templates can be altered by adding and removing of start parameters, return parameters and permission settings.
- **Req 11**: Change and optional change operations on single or multiple process instances and process templates are available as well as a possibility to group operations and applying them atomically
- **Req 12**: All or selected process instances can be migrated, when their process template's process model is changed

2.1.4 Analysis stage

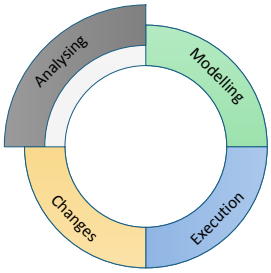


Figure 2.5: Analyzing

Workflow Management Systems should log the events, that lead to a process instance's termination. The activities and control flow decisions, input and output and the activities temporal aspects are a key information when performance reports are compiled by the **analysis stakeholder**. The WfMS needs to aggregate this information in a fashion, that clearly states, which activities are predecessors and successors.

We require the WfMS to link protocoled information across the entire range of entities: From activity instances and templates, over process instances to process templates to the staff member from the organization model, that executed the activity.

Thus, we formulate the following requirement:

- **Req 13**: Generate and provide access to log information

2.2 General requirements

In this section, general requirements that cannot be classified inside the process's life cycle are discussed. They span across all four stages of the process life-cycle or are stemming from the very nature of using a language as an interface for WfMS.

2.2.1 Portability

A large number of different operating systems and architectures are in use in companies. Prominent examples for operating systems are: Windows, Mac OS X, Linux, iOS, Android and Windows Mobile. Furthermore, a part of these operating systems can be deployed on a number of architectures, such as: the x86 architecture, ARM, PowerPC, Sparc and many more. Generally, software is developed for one or more target architectures operating systems, possibly by use of a framework.

When frameworks are used, they are - as software generally is - implemented for a number of architectures and operating systems. The more frameworks a program uses, the more it becomes dependent on the framework's supported platforms.

We require however a WfMS to be accessible from any device, regardless of the operating system, as long as it supportes a means of communication with the WfMS.

Thus, we state the following requirement:

- **Req 14**: Access to the WfMS may not exclude certain architectures, platforms or operating systems per design

2.2.2 Exchangeability

One of our primary motivations is the exchangeability of WfMS or components of WfMS (including activity programs). Therefore, a WfMS client and a WfMS server must be exchangeable by another server, if they both support a common interface.

We thus formulate the following requirement:

- **Req 15**: WfMS servers and WfMS clients are exchangeable

2.2.3 Concurrency

WfMS are generally used by more than one stakeholder at a time. A WfMS must therefore support concurrent operations. An interface may require to aid the WfMS in executing concurrent operations:

- **Req 16**: WfMS servers must support concurrent operations, the interface should aid the concurrent execution

2.2.4 Data serialization

Activity instances require input data and generate output data. The input is acquired by the activity stakeholder during check-out, the output returned during check-in. This data thus needs to be transported.

Serialization is the conversion of non-serial data, i.e. data that spans over multiple entities and attribute, into one serial block of data. It is commonly used when transporting data over a network, due to the complexities and network latency that would come with a per entity / attribute access to the data.

The language requires a robust and human readable data representation. Although binary data should be allowed, it is expected, that most data is stored in predefined basic data types and user defined data types that can be read in the form of text.

Numbers, dates and time-spans ought to be represented in a human and computer readable fashion. Hierarchical organized data should be represented in a way, that allows the human reader to implicitly understand the underlying data structure.

Thus, we formulate the following requirement:

- **Req 17**: Serialized data is required to be human and computer readable

2.2.5 Exceptions

We define **Exceptions** as the carrier of failure information for failures in the execution of processes and verification issues that the WfMS generates, when it cannot apply changes due to soundness restrictions.

Thus, we formulate the following requirement:

- **Req 18**: Exceptions express failures and errors. They contain information about the origin of the failure and the input data that lead to the failure

2.2.6 Unambiguousness

The language's semantic has to be unambiguous. An operation has exactly one meaning:

- **Req 19**: The WfQL's semantic is unambiguous

2.3 Evaluation of current standards

After listing the requirements, we searched for existing standards that could satisfy them. We considered the Web Services Business Process Execution language (WS-BPEL) [Me08], the Business Process Query Language (BPQL) [MS04], the Workflow Management Facility Specifications (WfMFS) [WfMFS], the XML Process Definition Language (XPDL) [XPDL] and in conjunction with the XPDL, the Business Process Modeling Notation (BPMN) [BPMN].

We split these standards into two groups: Firstly, WS-BPEL, WfMFS and BPQL that specify, how to execute actions on and how to gain information from a WfMS, and secondly, XPDL and BPMN that standardize the representations of workflow specific information like process models.

The **WS-BPEL** can be used to execute workflows via Web Services. The concept is interesting, however does not include our requirements from the analysis stage and the change stage. The **BPQL** is solely concerned with the analysis of past run processes and does not cover requirements from the modeling, execution and change stage. The **WfMFS** by the Object Management Group is simply put, a RPC standard for access of WfMS and does not concern itself with the concurrency requirements or organization models. It is using the Common Object Request Broker Architecture (**Corba**) [CORBA] which would introduce further framework dependencies. Though being closest to our requirements, the WfMFS disqualifies itself as a solution by being, simply put, too generic to be of practical use. In addition, according to [He06], licenses for commercial Corba implementations are typically expensive and have several technical issues.

The second group of standards, **XPDL** and **BPMN** cannot satisfy any of our requirements. They are only interesting in so far, that they represent process models. This representa-

tion could be used as a serialization method of process models, if they would match the underlying process meta model.

We failed to find any suitable, existing standard protocol / language that would satisfy our requirements. Therefore, we have no other choice but to design our own language.

2.4 Summary

This chapter presented a number of requirements towards both a WfMS and an interface for WfMS. We evaluated a number of standards against the requirements and came to the conclusion, that none of them match our requirements. Therefore, we start making assumptions and design decisions on a WfMS and its interface in the following chapter in order to design an interface, that matches our requirements.

3 Assumptions and Design Decisions

In sec. 2.3 we evaluated a number of existing standards. We came to the conculsion, that none of them satisfies our requirements from chapter 2. We therefore decided to design an interface language from scratch. However, a language may only communicate information based on a common meta model. We thus state assumptions and design decisions in this chapter and argue at each decision, why a model was chosen, and if there are viable alternatives.

We availed ourselves of solutions from the following works: [Re00], [Fo09] and [Be05] and put them into context.

3.1 Staff assignment rules and device restrictions

Access control, staff assignment and its enforcement is required. Req 5 outlines the problematic. In this section, we discuss this and related challenges and formulate the problem more precisely.

We prerequisite, that companies have a sort of organization structure like departments, project groups and individual work positions.

With a multitude of devices possibly able to use the WfMS through an interface, an additional restriction comes to mind: Restricting abilities not only per ability of the stakeholder involved, but also per device.

We found a number of meta models that might solve the staff assignment problematic: Unix-style operating systems use a permission management based on users and groups. Resources belong to a user and to a group. Possible permissions include read, write and execute. A user may for example be allowed to write and read his file while the group may only read it. This system is often too simplistic for complex multiuser systems and is often complemented by **Access Control Lists** (ACL) [POSIX-1003.1e]. ACLs allow for a more

fine granulated permission management, but do not hierarchically organize the structure. The latter is paramount, when reflecting an organization's hierarchy. Therefore, ACLs are not suitable as basis for staff assignment rules.

When evaluating [Be05] we found an organization model that suites our requirements. It is well documented, shows the relationships between different entities within an organization's hierarchy and imposes sensible constraints. When compared to ACLs, it is more complex as a meta model. However, it allows for a simpler structure when actually using it by predefining commonly found entities in organizations such as organization units, project teams, positions that would otherwise have been modeled with a generic meta model, that would not impose constraints and would allow unsound constructs.

We therefore make the following assumption:

- **Assumption 1**: The organization model from [Be05] is used as meta model for the organization hierarchy

The target platforms for the WfQL include embedded systems and cell phones. These devices have obvious shortcomings in displaying information and accepting human user input. We think it sensible, to offer a model to restrict the abilities of these devices so that no device, that is not capable of actually executing an activity can accept it.

For example, an intelligent door may execute the activity of opening itself. However, it will probably not have a keyboard, and if known, that it has no keyboard, the ability of inputting text should be revoked from the device.

We thus make the following assumption:

- **Assumption 2**: A device restriction model is used, that revokes abilities for devices and groups of devices

3.2 Transactions

Many changes to entities in a workflow management system should only be applied in conjunction. A change may depend on another change for the WfMS to get into a sound state.

Per Req 5 only sound models may get persisted. A verification is however not at all times possible directly after the change was applied – it might require changes in other

entities which in turn cannot be changed without modifications to the original entity. Such a deadlock cannot be broken, if each change is verified and rejected separately.

Therefore, the WfQL should offer a construct, which allows modifying many entities at the same time. Additionally, concurrency has to be considered. If more than one stakeholder operates on a WfMS at the same time, we expect them reading and writing overlapping data.

Thus we require the following attributes:

- **Atomicy** – Group operations together and apply them as one
- **Consistency** – Verify that changes are sound
- **Isolation** – Premature writes may not cause dirty reads[1]
- **Durability** – Persistence after operations are accepted

Thus we can formulate the following requirement:

- **Assumption 3**: Offer transactions with ACID properties

3.3 Block structure

In Req 5 we required, that process models be sound. We thus require either a process model meta model, that allows the verification of the soundness of the process model, or that it is sound per design. If the latter can be obtained without imposing constraints, it would be the optimal choice.

As stated in [Re00], a block structured process model with the therein stated properties is always sound and correct. Figure 3.1 shows an Adept style process model example.

Although a number of constraints are superimposed on the process model, we failed to find process models, that cannot be modeled via the Adept meta model from [Re00]. We therefore make the following design decision:

- **Assumption 4**: The process model uses the block structure from the Adept2$_{Flex}$ meta model from [Re00]

[1] Solves overlapping read and write problematic when multiple stakeholders access the WfMS concurrently

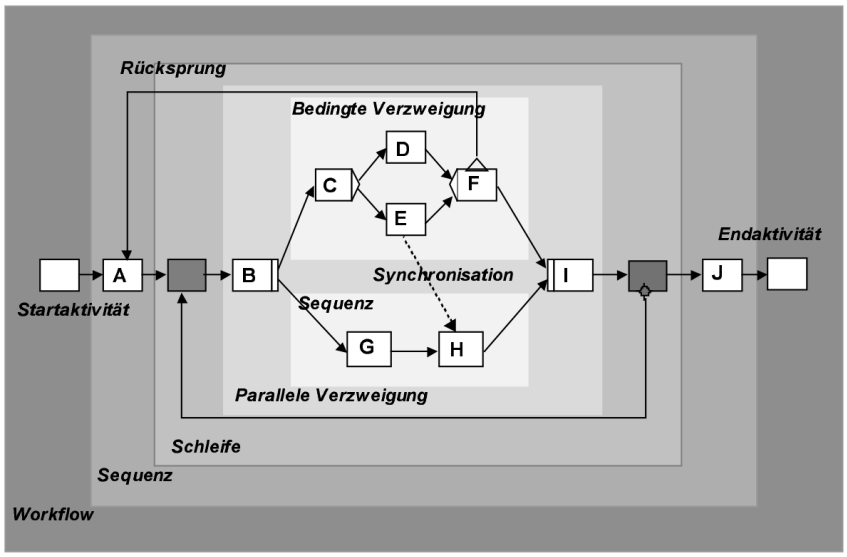


Figure 3.1: Block Structure from [Re00]

3.4 Data flow

Data flow is a required property of the WfMS and needs to be modeled with help of the WfQL.

There are two opposing approaches to model data flow that need to be discussed:

1. **Direct data flow** describes, which return parameters of activities become input parameters of successive activities.
2. **Using data containers** in the context of a process, to store return parameters and to use data container as input parameters for activities.

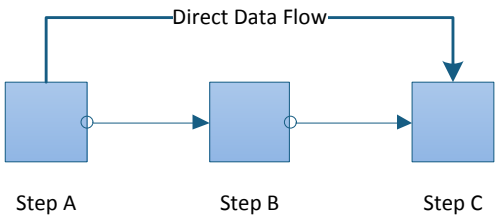


Figure 3.2: Direct Data Flow

When inspecting a process instance, the direct data would not offer means to inspect the current state of the process instance – the analyst stakeholder would be required to inspect the return parameters of previously ran activities and follow the modeled data flow to understand the process's state. In addition, moving activities inside a process becomes more complicated when ac-

tivity's output is mapped directly to input, which requires additional validation that becomes increasingly complex. Due to this complexity, we strongly lean towards using data containers.

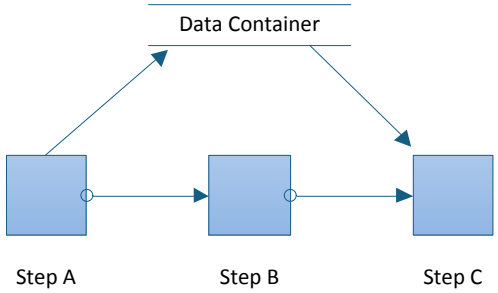


Figure 3.3: Data Containers

Using data container is problematic when activities in parallely executed branches of the control flow read from and write to the same data container. Thus synchronization is required to guarantee the order of reads and writes. [Re00] introduces **synchronization edges**[2] that require the node, that a synchronization edge is originating from, to wait for the target of the edge. Therefore, solving the order problematic.

We thus state the following assumptions:

- **Assumption 5**: Data containers are used to store the output of activities and read as source of input for activities and control flow decisions
- **Assumption 6**: When the read / write order of data containers is not clear by the structure of the process model, synchronization edges must be used to order read and writes

3.5 Decision Logic

A process's execution engine may need to decide which control path to choose. This is an integral part of the control flow, and thus, is part of the modeling and covered by the WfQL.

We see two options for solving this problem:

1. **Activity Based decision logic** that uses activities to decide, which control path is chosen.

[2] ger. Synchronisationskanten

2. **Decision Predicate based decision logic**, that labels control edges with rules, that can be evaluated and thus let the execution engine decide, which control path is chosen.

Activity based decision logic moves the actual decision of the continuance of the control flow out of the model and out of the WfMS's context. This is undesirable because of several aspects: For one, the actual implementation of the activity is out of the scope of the model and thus can not be tested when evaluating the model alone. Second, the activity actually needs to be executed, which implicitly requires a client to run at all times. Third, the activity client may only have limited access to the scope of the process instance – namely, the scope of any activity.

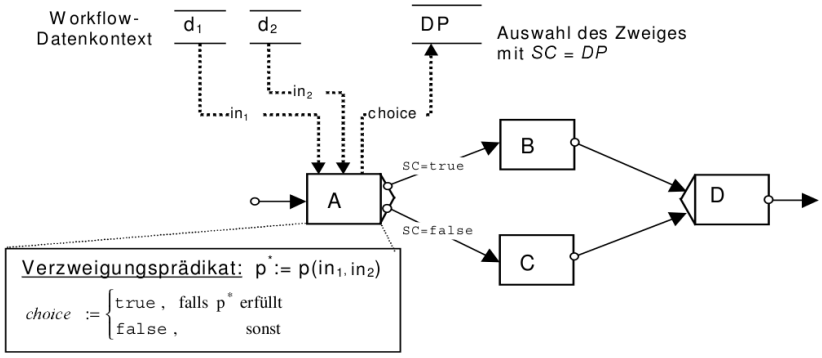


Figure 3.4: Decision Predicates from [Re00] (Figure 3-4)

The decision predicate based decision logic does not have the shortcomings of the activity based decision logic. It allows for the decision logic to stay within the model, and executable by the process execution engine. The decision predicate can be formulated with the WfQL and thus allows for a much broader spectrum of information available to the decision than just the data containers of the process instance. All aspects of the current process instance – and other instances – can be considered.

We therefore formulate the following assumption:

- **Assumption 7**: The decision predicate from [Re00] is implemented

With the decision logic being an part of the model, possible paths can be tested and thus evaluated, if a process model is deterministic. A short discussion on this topic can be found in sec. 7.1.

3.6 Data types and constraints

With data containers storing information, the output of activities needs to be verified in order to ensure, that when data containers are used as input to activities and decision predicates, they contain data of a certain type.

When the data is verified at the time of the writing into data containers, failures to comply to the modeled types get caught at the source.

Req 3 states, that only sound control flow constructs should be accepted. Therefore, a process instance needs to terminate, regardless of the activity instances return variables. It may abort due to errors, but it never may get in a state, where no continuing control path can be chosen.

Data containers may contain a wide range of data which is only limited by the data type. In conjunction with **Req 2** we think it necessary, to further limit the value that data container can contain in the form of data constraints.

- **Assumption 8**: Allow definition of new data types by use of existing ones
- **Assumption 9**: Offer an aggregation data type to store multiple elements accessible by name
- **Assumption 10**: Offer an aggregation data type to store multiple elements of the same type
- **Assumption 11**: Allow definition of constraints on data container definition in process instances and data type definition

3.7 Change Model

Due to Req 3 we are bound to verify the integrity of process models. With Assumption 4 a block structure was superimposed on the process model which is an additional constraint to the meta model.

We are going to make use of the change patterns presented in the technical report [WRS08] that shows a number of change patterns for block structured process models that satisfy our requirements and assumptions.

Thus we formulate the following assumptions:

- **Assumption 12**: Changes to process models are executed via a subset of the operations defined in [WRS08]

3.8 Logging and the general entity relationship meta model

Per Req 13 we stated, that we need to aggregate activity execution per executioner stakeholder, process template, process instance and activity template. Thus, we require a metamodel to log any progress made that references the entities that are related to the logged information.

For example, the log information of an activity instance execution is related to the activity template, that the activity instance is based on and the process model, which in turn is related to a process instance which in turn is based on a process template. Additionally, the staff member / stakeholder that executed the activity instance is related to the activity instance execution.

Figure 3.5 shows a general entity relationship diagram that the WfMS is required to implement. A logging entity stands in relation to the entity that triggered its creation. Thus, with this relation, it is possible to afterwards analyze all depending entities – and to vice versa, aggregate logging entities that were created for entities.

- **Assumption 13**: The WfMS implements an entity relationship model similar to Figure 3.5

3.9 Summary

This chapter presented the meta models of a WfMS. They form the basis of the following definition of the WfMS model in chapter 4 and the language definition in chapter 5.

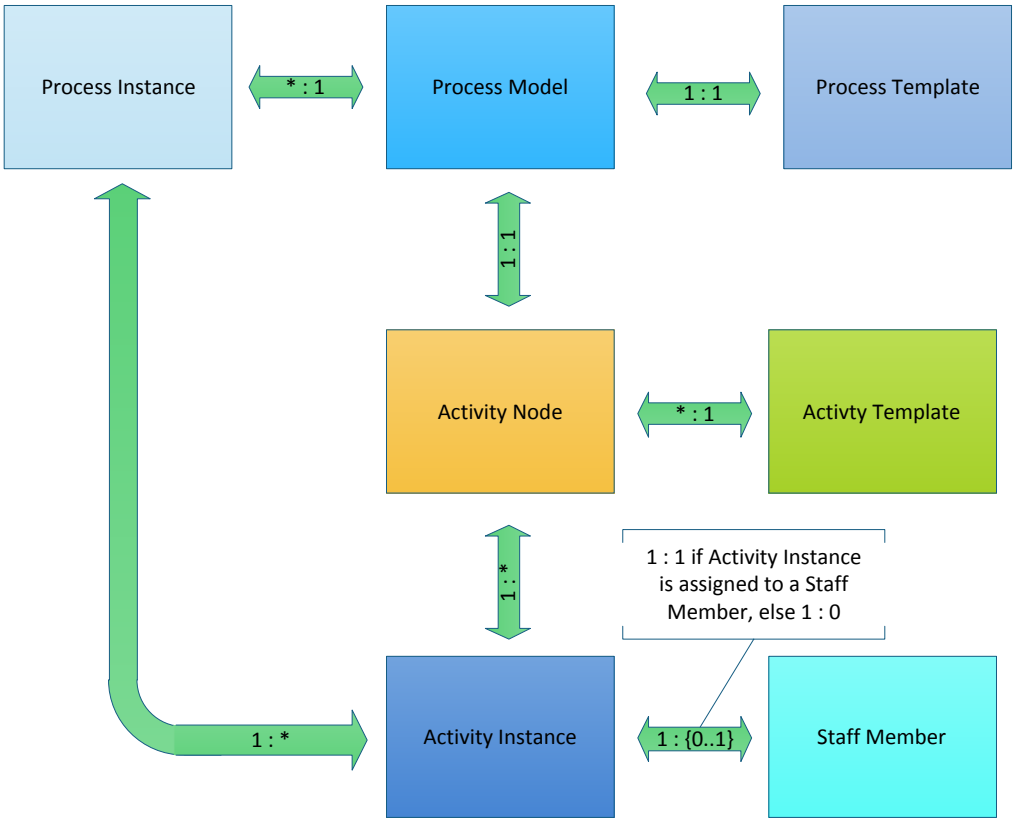


Figure 3.5: Entity Relationship of the WfMS

4 Realization

This chapter presents a meta model for WfMS based on the requirements and assumptions from chapter 2 and 3. All WfMS implementing an interface for the WfQL have to comply to this model in order to be fully compatible. It is expected, that a WfMS internally represents its entities and data in other schemes, and that the interface translates operations from the WfQL into the specific WfMS's implementation.

We first present the organization model and the device restriction model with its entities and relations, followed by the data types and constraints. Exception complement data types by offering a means to store failure information. Based on these fundamental declarations, the process meta model is realized.

4.1 General

We split the WfMS in three component groups: **storage**, **application** and **interface**. Illustrated in Figure 4.1 is the relationship between these groups. The application component interacts with storage, the interface component with the application component.

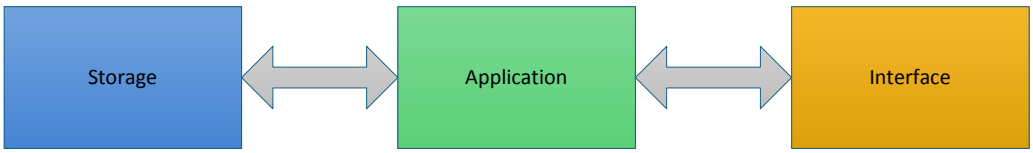


Figure 4.1: Components

In this chapter, we specify the application component's behavior and offers in part models for the storage of WfMS specific information.

The interface component is split into two distinct parts: The communication part and the parser. The latter makes use of the language definition from chapter 5. For an example

implementation of an interface component, the prototypical server implementation shows a multi-threaded TCP/IP socket server, that uses an (exchangeable) parser, that implements the WfQL.

4.2 Organization Model and Restriction Model

Assumption 1 states, that the organization model from [Be05] should be used. Assumption 2 , that a device restriction model should be implemented. This section puts the organization model and a device restriction model into the context of a WfMS realization.

Organization Model

The organization model from [Be05] is a meta model for picturing organization structures. It stores this information within entities, that stand in relation to other entities of the model. Figure 4.2 shows the entity relationship diagram based on [Be05], that we use to store the information.

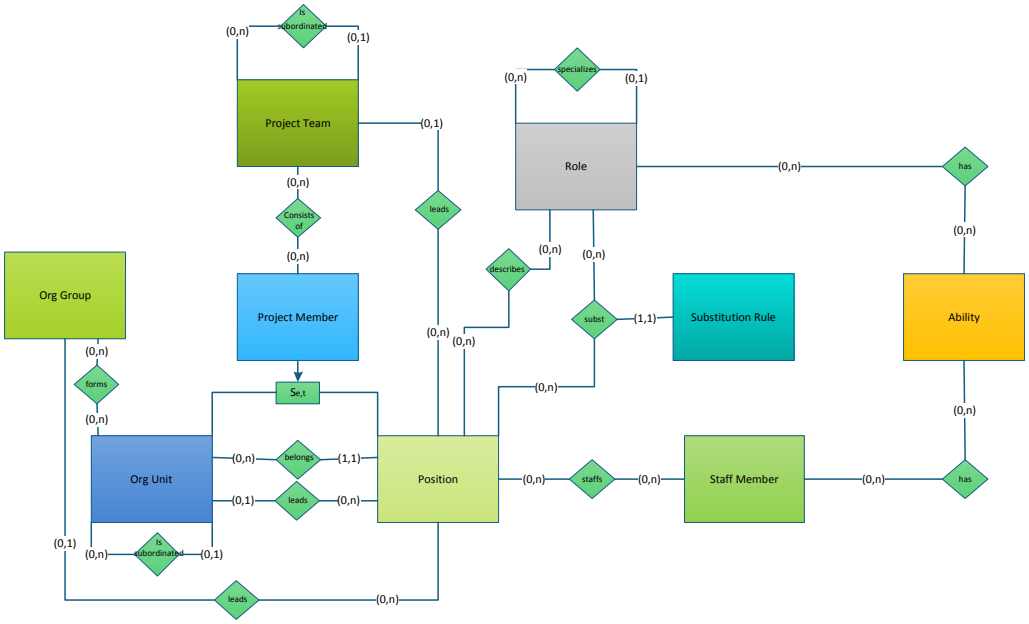


Figure 4.2: Organization meta model based on [Be05]

Staff Members are representing human beings or computer programs. With **abilities**, certain operations are allowed. A group of abilities is aggregated in **Roles**. Staff members have roles, and **Positions**. The latter aggregates roles. The **Substitution Role** expresses (possibly time limited) substitution Positions, so that in case of unavailability of a Position, another Position can temporarily gain access to the Roles of the substituted Position. **Org Units** form **Org Groups** and subsumes Positions. One Position is leading an Org Unit, an Org Unit has any number of Positions. **Project Teams** are lead by a Position, and consist of Project Members, which are either Org Units or Positions. Project Teams can be subordinated under other Project Teams.

4.2.1 Organization Model Entity Definition

Based on [Be05] and Figure 4.2 we present the entity definition for use in WfMS. Many-to-many relationships are declared on the entities of both sides of the relationship and numbered with roman numbers to identify them. This numbering reused in the language definition to reference the relationships.

Org Group

Organization groups have one lead position and group organization units.

Organization groups are represented by the following tuple:

name	lead_position
name	Unique identifier for organization groups
lead_position	References the position, that leads the organization group (optional)

Organization groups may have many organization units, which may be subordinated to many organization groups. Therefore, we formulate this mapping:

- many to many relationship with organization units (I)

Org Unit

Organization units subsume positions. They are lead by a position. They can be subordinated to another organization unit. They may have project teams.

Organization units are represented by the following tuple:

name	higher_org_unit	lead_position
name	Unique identifier for organization units	
higher_org_unit	Reference to a superordinated organization unit (optional)	
lead_position	References the position, that leads the organization unit (optional)	

Organization units may be part of multiple (or none) organization groups and may have multiple (or none) project teams. We thus formulate the following mappings:

- many to many relationship with organization groups (I)
- many to many relationship with project teams (II)

Project Team

Project teams consist of positions and are part of organization units. They are lead by a position and can be subordinated to one other project teams.

Project teams are represented by the following tuple:

name	higher_project_team	lead_position
name	Unique identifier for project teams	
higher_project_team	Reference to a superordinated project team (optional)	
lead_position	References the position, that leads the project team (optional)	

Project teams may be part of multiple (or none) organization units and may consist of multiple (or none) positions. We thus formulate the following mappings:

- many to many relationship with organization units (II)
- many to many relationship with positions (III)

Position

Positions describe (none or many) Roles. Staff members may have positions. Project

teams consist of positions. The substitution rule expresses the (temporary) access to the roles of another position.

Positions are represented by the following tuple:

name	org_unit
name	Unique identifier for positions
org_unit	References the organization unit, this position belongs to

Positions may be staffed by multiple (or none) staff members. Positions are described by multiple (or none) Roles. Positions may be part of multiple (or none) project teams. Positions may substitute other position via substitution rules . We thus formulate the following mappings:

- many to many relationship with staff members (IV)
- many to many relationship with roles (V)
- many to many relationship with project teams (III)
- many to many relationship with substitution rules (VII)

Role

Role group abilities. Positions may have roles. Multiple roles can be (temporarily) substituted per substitution rule.

Roles are represented by the following tuple:

name	higher_role
name	Unique identifier for roles
higher_role	Reference to a superordinate role (optional)

Positions are described via their multiple (or none) roles. Roles have multiple (or none) abilities. A substitution rule has multiple (or none) roles. We thus formulate the following mappings:

- many to many relationship with positions (V)
- many to many relationship with substitution rules (VI)
- many to many relationship with abilities (VIII)

Ability

Abilities describe the competences of the related entities. Staff members can directly reference their abilities. Indirectly, staff members can have abilities via their positions, that have roles, that have abilities.

Abilities are represented by the following tuple:

name	
name	Unique identifier for abilities

Roles have multiple (or none) abilities. Staff members have multiple (or none) abilities. We thus formulate the following mappings:

- many to many relationship with roles (VIII)
- many to many relationship with staff members (IX)

Staff Member

Staff members describe employees of an organization. They staff positions and have abilities.

Staff members are represented by the following tuple:

name	
name	Unique identifier for staff members

Staff members staff multiple (or none) positions. Staff members have multiple (or none) abilities. We thus formulate the following mappings:

- many to many relationship with positions (IV)
- many to many relationship with abilities (IX)

Substitution Rule

The substitution rule describes temporary access to roles for positions, that are thus gaining access to roles of other positions, that are substituted this way.

Substitution rules are represented by the following tuple:

name	start_date	end_date

name	Unique identifier for substitution rules
start_date	Describes the start date of the rule (optional)
end_date	Describes the end date of the rule (optional)

Substitution rules reference positions and roles. The referenced positions gain access to the referenced roles based during the timespan (between start_date and end_date) that the substitution rule defines. We thus formulate the following mappings:

- many to many relationship with positions (VII)
- many to many relationship with roles (VI)

Summary

The organization model's entities and their relation was presented in this subsection. This scheme is reused in the language definition. It is expected, that the WfMS implements this scheme (or translates its scheme into a compatible one) for WfQL interfaces.

4.2.2 Device Restrictions

Device restrictions formulate, how abilities are revoked from staff members that are using specific devices according to the entity relationship diagram in Figure 4.3. The WfMS determines which abilities a staff members has with the help of the organization model. Afterwards, it revokes abilities based on the used device and the device restriction model.

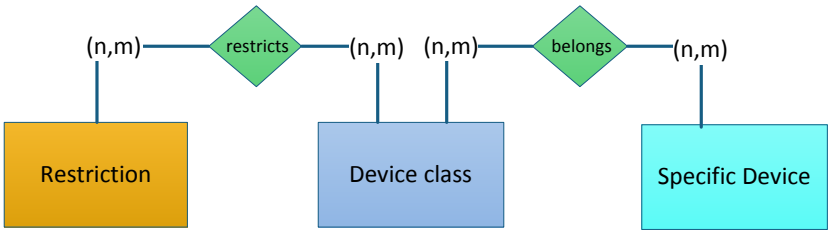


Figure 4.3: Device restriction model

4.2.3 Restriction Model Entity Definition

The following paragraphs define the entities of the device restriction model and their relationships as illustrated in Figure 4.3.

Restriction

Restrictions reference an ability from the organization model. A restriction expresses, that the referenced ability is not available. Device classes map restrictions, to revoke abilities.

Restrictions are represented by the following tuple:

name	ability
name	Unique identifier for restrictions
ability	References an ability

Restrictions are used by multiple (or none) device classes. We thus formulate the following mapping:

- many to many relationship with Device classes (I)

Device class

Device classes group specific devices. A device class can have restrictions.

Device classes are represented by the following tuple:

name	
name	Unique identifier for device classes

Device classes refer multiple (or none) restrictions and multiple (or none) specific devices. We thus formulate the following mappings:

- many to many relationship with Restrictions (I)
- many to many relationship with Specific Devices (II)

Specific Device

The specific device entity represents a device that is grouped with similar devices in device classes. A device can be part of many device classes.

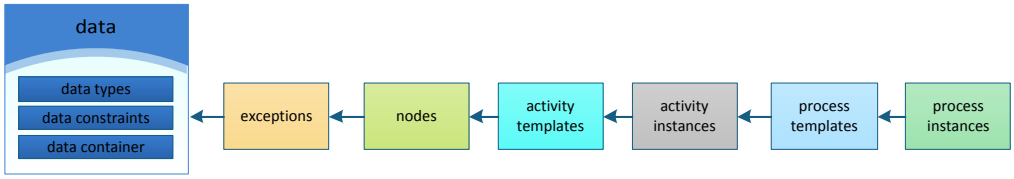
Special devices are represented by the following tuple:

name	
name	Unique identifier for specific devices

Specific devices refer multiple (or none) device classes that they are part of. We thus formulate the following mapping:

- many to many relationship with Device classes (II)

4.3 Data representation



A WfMS stores a number of different types of data. **Data containers** are used to store data. They are typed, so their values and attributes can be validated.

4.3.1 Data types

Using data types means, that the data is of a certain type and follows a specific format. The following data is typed:

- Data Container
- Start / Return Parameters of:
  - Process Templates / Instances
  - Activity Templates / Instances

We differentiate types into two classes: Basic Data Types that are predefined and User Defined Data Types, that are defined based on the basic data types.

The following basic data types are predefined:

integers	positive and negative integers
numbers	floating point numbers
text	text of unlimited length
timespan	time spans with microsecond precision
date	date with second precision

User defined data types use either preexisting user defined data types or basic data types in aggregating data types such as Maps, Structs, Lists and Sets.

Maps map a typed key to a typed value, **Structs** define keys by name and a typed value. **Lists** and **Sets** subsume data containers of the same type. Sets are unordered, Lists ordered.

Exceptions are a special case of structures with additional possible key - value pairs. They contain the exception's name, the originator entity, a stack trace and a message that contains a short error description.

This part overlaps with the language definition, as data types are used in the interface and a notation is introduced that is capable of representing all data types used. Additional information can be found in sec. 5.2.3, Table 5.2 and Table 5.1.

4.3.2 Data constraints

Constrains per basic data type are:

- Numbers and time-spans: Range constrains
- Dates: Date constrains
- Text: maximum and minimum length in characters, must contain specific words or phrases
- Binary data: maximum and minimum length

Two constraints levels are possible:

1. **Data type level:** Bound to the data type, all elements of this type have to follow this constraint and can only be absolute.
2. **Process level:** Bound to the specific data element of a process. They are specified at the declaration level of data containers.

4.3.3 Data containers

Data containers are used to store information about the current state of a process in the process's instance. The process model declares data containers with a type and possibly with constraints (see sec. 4.3.2).

Start parameters of process models are mapped to data containers, return values of activities are mapped to data containers, input parameters of activities and the decision predicates are mapped from data containers of process models.

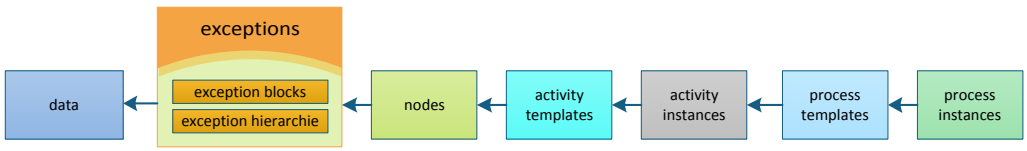
A data containers initial value is the NULL pointer if possible, a numeric zero, an empty text, a timespan of zero or, in case of dates, the first date possibly represented. This trade-off has been made in order to accommodate the many possible development platforms and languages that may be used in the implementation.

Data containers definitions are declared by the following tuple:

process_model	name	data_type	process level constraints

process_model	Reference to the process model, the data container is part of
name	Name of the data container
data_type	Reference to the data containers data type
process level constraints	defines the data constraints (instance level) on this data container

4.4 Exceptions



Exceptions are encapsulated error reports that the execution engine generates when errors occur that cannot be solved at the point of origin. They contain an exception type, a stack trace with references to all entities that the exception passed through, and a reference to the origin. They are typed as data type within the type definition of the WfQL and can be used as data containers of this type in every respect: Their attributes can be accessed via decision predicates, they can be used as input parameters of activities, they may even be returned as return parameters of activity instances.

Exceptions are represented by the following tuple:

exceptionType	stackTrace	user defined fields

exceptionType	Name of the exception
stackTrace	list of pairs of entity type, reference and action
user defined fields	per ExceptionSubType declared fields

A stackTrace for a failed Activity Instance, whose stakeholder threw an exception could for example look like this:

- (`PROCESS_INSTANCE`, 11, START)
- (`ACTIVITY_NODE`, "Accept Orders", STARTED)
- (`ACTIVITY_INSTANCE`, 5, INSTANTIATED)
- (`ACTIVITY_INSTANCE`, 5, CHECKED-OUT)

By use of the stackTrace, it is possible to determine the path that lead to the Exception. We however make no claims, on how detailed the stack trace should be. The only definition we do make, is, that the entity involved must be named per the entity name (see language definition, `entitySet`, sec. 5.2.5) and the identifier to be according to the the language definition.

With the data type definition that follows in chapter 5, it is possible to define user defined exceptions. We predefined a number of exception that cannot be changed or overwritten, that the WfMS's execution engine triggers in case of failure:

Exception name: Data Type or Constraint Violation
Created when trying to write data into a data container that does not match the target's type. Example: Writing a Text into an Integer data container. If constraints were violated, the constraint's definition is written in 'violated constraint'. If the type was mismatched, 'violated constraint' remains null.

The exception can be represented by the following tuple:

cause	Reference to Entity
expected type	Typename
received type	Typename
violated constraint	Constraint or NULL

Exception name: Control flow decision failure

This exception is created when a control flow decision predicate did not evaluate a sound control path decision. Example: Two out of three XOR Decision Predicates returned true. Only one was expected.

Note: With data constraints and data types, the possibility of control flow decision failures was minimized. It is however still possible, and we do not make assumptions on how intensively the WfMS checks the model for possible failures. Hence, it was necessary to envision a failure handling for malformed decision predicates.

The exception can be represented by the following tuple:

cause	Reference to Entity
expected # of selected control flows	Integer
actually selected # of control flows	Integer
list of selected nodes	list of decision predicate and reference to node pairs
list of not selected nodes	list of decision predicate and reference to node pairs

Exception name: Activity timed out

Activity nodes can set a maximum execution time of its activity instance, and a maximum waiting time for its activity instances to wait for a stakeholder to execute it.

Note: If the activity instance is executed by a stakeholder when the exception occurred, the stakeholder is notified that the instance was terminated.

The exception can be represented by the following tuple:

cause	Reference to Activity Instance
elapsed time	time of type timespan
type	execution time or waiting time

Exception name: Staff Assignment Failure

A stake assignment query did not select any staff members. Hence, an activity instance has no stakeholders that are allowed to execute it. The exception before the activity instance is created.

The exception can be represented by the following tuple:

cause	Reference to Activity Node and Process Instance

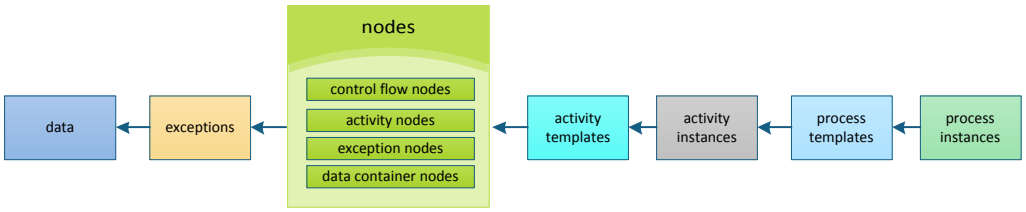
Exception aborting control flows

Exceptions can abort running control flows in other branches of the control flow. It is a direct result of the nature of exceptions: Only one exception can be handled per block. If an exception is thrown by an exception handling block to an outer block, all control flows within the block have to be aborted and the data containers wiped.

4.5 Process models, templates and instances

Processes can be visualized and understood as graphs with a start and an end with blocks and activities in between. This section is describing different types of nodes, that are used to model blocks, decision logic and failure handling.

4.5.1 Node and Block types



Process models contain different node types. All types have common properties and special ones, depending on the node's characteristics. A node can only be part of one process model. When not otherwise specified, a node may only have one predecessor and one successor. A block is a special case of a node and can encompass other nodes.

In the execution materialization of a process template, the process instance, nodes can have the following states:

1. waiting
2. ready
3. active
4. finished

A node in waiting-status waits for its execution. Ready means, that the previous nodes were executed and that this node should be executed as soon as possible. Active, that it is currently being processed and finished, that this node was processed and the next node is not in the state "waiting" any more.

No extra failure status exists. A node that encountered an error changes its state to "finished".

Activity Nodes

Activity template are referenced from nodes of type **activity node**. In such a reference, the required input parameters of the activity templates and the return parameters are mapped from and to data containers.

Staff members are selected with a staff assignment query (SAQ). This query, and the activity template's required abilities are used in combination to filter the staff members.

An activity node can be represented by the following tupel:

activity_template	input mapping	output mapping	staff assignment	status
waiting timeout	execution_timeout	SAQ		

activity_template	reference to an activity template
input mapping	maps data containers to input parameters
output mapping	maps output parameters to data containers
staff assignment	predicate, that select staff members from the organisation model
status	waiting, ready, active or finished
waiting timeout	NULL (none) or a timespan. maximum waiting time before a timeout exception is issued
execution timeout	Analogue to waiting timeout, starts its count after the activity instance was started
SAQ	staff assignment query

Status waiting, ready and finished are analogue to the common node specifications. When switching from ready to active, an **activity instance** is created with the input parameters mapped from the process instances context. The node switches into state **active** and remains in it, until the activity instance terminates. If terminated successfully, the return parameters are written to the corresponding data containers. If not, an exception is created. In both cases, the node switches to status **finished**.

Blocks

Blocks have a name, a type, and possibly sub control flows, lists of embedded blocks or nodes, exception blocks, data containers and data container mappings. A block can always be used in place of a node.

Serial Block

The serial block encompasses one or more nodes, that are executed serially.

It can be represented by the following tuple:

name	data containers	input-mapping	output-mapping	nodes	exception-blocks

An example is shown in Figure 4.4. The serial block is named “SerialBlock” with a data container named “Data Container”, with its Nodes “A” and “B” and an exception block. Per definition, the serial block states, that its nodes are executed serially. So, node “A” is executed and the execution continued with node “B”.

If an error occurs, the WfMS looks for an exception catch block. In the example, one is modeled that executes node “D” and then throws the exception to the next higher block. Alternatively, an exception jump node could have been modeled, that would point to a node within the nodes-list of the “SerialBlock”.

Data Containers can be mapped from outer data containers, defined in a hierarchically higher block to inner data containers, like “Data Container” in this case. Values of inner data containers can analogue be mapped to outer data containers. The mapped data containers are copied over when the block terminates and starts respectively.

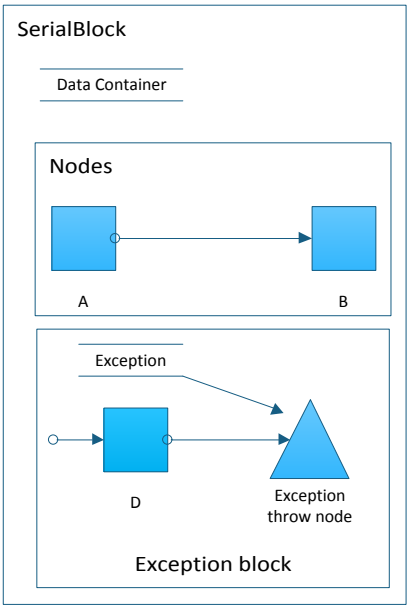


Figure 4.4: Serial Block

Control Flow Blocks

While a serial block aggregates multiple nodes, control flow blocks can model the execution of parallelly executed nodes and the selection of one or multiple nodes based on decision predicates.

A control flow block makes use of different split and join mechanisms. We use XOR and AND Split and Joins from the Adept Meta Model and supplement it with OR Split and Joins. The type of split and join is named in the declaration of the control flow. Depending on the split, the encompassed nodes in sub control flows are labeled with a decision predicate.

A control flow block with a split of type OR and XOR can be represented by the following tuple:

name	split-type	join-type	data containers
input-mapping	output-mapping	list of dp : node pairs	exception-blocks

The *list of decision predicate : node pairs* expresses the possible sub control flows.

Control flow blocks with a split of type AND can be represented by this tuple:

name	split-type	join-type	data containers
input-mapping	output-mapping	list of nodes	exception-blocks

Due to the nature of an AND split, no decision predicate is necessary.

The following table lists the possible splits and joins and their behavior:

XOR-Split	Splits the control flow into one or more sub control flows, from which only one is selected determined by a decision predicate that is expressed via the WfQL.
XOR-Join	Joins the control flow. The first sub control flow arriving at the XOR-Join aborts all other, possibly running control flows.
AND-Split	All sub control flows are activated. No selection possible.
AND-Join	Waits for all sub control flows to arrive.
OR-Split	Any number of sub control flows can be selected, determined by a decision predicate that is expressed via the WfQL.
OR-Join	Waits for one sub control flow to arrive and then continues. Other sub control flows are not aborted and end with the OR-Join.

Loop Blocks

Loop Blocks encompass a control flow, that is executed one or many times. Loop blocks are similar to sequential blocks with an additional decision predicate, that is evaluated after list of nodes was executed. If the decision predicate evaluates to true, the list of nodes is executed again. Else, the control flow continues.

Thus, loop blocks can be represented by the following tuple:

name	data containers	input-mapping	output-mapping
inner node	exception-blocks	decision predicate	

Dynamic parallel execution

The dynamic parallel execution supports the execution of any number of parallelly executed control flows depending on the contents of data containers. It is expresses through dynamic parallel execution blocks, or shorthand, dynpar blocks.

A dynpar block requires a data container of type set or list as input parameter. It must contain a data container which must be of the type of the elements of the input parameter set / list. The dynpar block defines a special mapping from the input parameter to the data container. For each element of the input parameter, a new control flow is created with the element residing in the specified data container.

In addition, a data container can be specified, that is aggregated to a set and can be mapped to outer data containers, thus allowing for return parameters.

The dynpar blocks can be represented by the following tuple:

name	data containers	input-mapping	output-mapping
node	exception-blocks	special-input-mapping	special-output-mapping

Exception blocks and special nodes

Exception blocks are defined in regular blocks. Their name reflects the exception's name that they handle. If an exception occurs within the block the exception block was specified in, the exception block is executed. According to the exception block's mapping, the exception is written into a data container.

For use in exception blocks, two special exception nodes are defined. The exception throw node throws a referenced Exception, that was previously stored in a data container. The catch block jump node behaves similar to an XOR-Split. The target node has to be the node that originally threw the exception or is located before it. It is therefore not allowed to skip a part of the workflow which could cause problems.

Exception block

An exception block is very similar to a serial block, with the additional constraints that is must end with either an exception throw node or an exception jump node. The exception XOR block allows for selection of one of many final throw nodes or jump nodes.

The following tuple represents exception blocks:

name	list of exceptions	node	data containers	special-input-mapping

name	name of the exception block
list of exceptions	a list of all exceptions that this block is capable of handling
list of nodes	list of nodes, that end with a exception jump node or exception throw node[1]
data containers	declaration of data containers
special-input-mapping	mapping of exceptions, source is the exception's name, target the local data container containing the exception. It is not allowed to use non local data containers.

Exception XOR Block

Exception XOR Blocks are very similar to XOR Split / XOR Join blocks. They contain decision predicate : serial block pairs. However, they require the last node executed within the block to be of type exception throw node or exception jump node to guarantee the integrity of the exception block. Their use is allowed only within exception blocks.

The following tuple represents exception xor blocks:

name	data containers	input-mapping	list of dp : node pairs

name	name of xor exception block
data containers	declaration of container declarations
input-mapping	regular block input mapping
list of dp : serial block pairs	a list of decision predicate and serial block pairs with the requirement, that the last node of the serial block node is either of type exception jump node or exception throw node

Exception throw node

An exception throw node can be expressed through the following tuple:

name	reference-to-exception-data-container

Exception jump node

Exception jump nodes jump to a node in the block, the exception block is part of, that was previously executed. We cannot per design prohibit falsely modeled jumps. The WfMS is expected to enforce this rule.

An exception jump node can be expressed through the following tuple:

name	reference to node

4.5.2 Process Model

Process models describe the model of processes. They are referenced in Process Templates and Process Instances. They can be expressed as a tuple with the following properties:

root node	modeling abilities	execution abilities	analysis abilities
root node	references the root node of the process model		
modeling abilities	required abilities for modeling		
execution abilities	required abilities for execution		
analysis abilities	required abilities for analysis		

The root node's input / output parameter mapping is used as the process's input / output mapping. The ability restrictions allow for a simple permission management.

4.5.3 Activity template

Activity templates contain all information necessary to instantiate activity instances based on the template and the start parameters. They also declare a set of return parameters and exceptions, that activity instances based on them can issue.

Required abilities state the abilities, that are necessary to execute an activity instance based on this activity template.

An activity template contains can be represented by the following tuple:

name	start_parameter	return_parameters	exceptions	required_abilities

name	Unique name
start_parameter	Data Type Definition of type Struct
return_parameters	Data Type Definition of type Struct
exceptions	list of exception names
required_abilities	list of names of required abilities

4.5.4 Activity instance

Activity instances express instantiated activities. Stakeholders check these activities out, process them, and check them back into the WfMS.

The activity instance can be represented by the following tuple:

start_parameters	return_parameters	activity_node	process_instance
activity_template	status	selected_staff_members	
creation_timestamp	checkout_timestamp	checkin_timestamp	

start_parameters	Values of Start Parameters
return_parameters	Values of the Return Parameters (initially null)
activity_node	Reference to the Activity Node (by name) that the instance is based on
process_instance	Reference to the Process Instance (ID) that the instance is based on
activity_template	Reference to Activity Template the instance is based on
status	ready-to-run, active, finished or failed
selected_staff_member	Staff members that can execute this Activity Instance
creation_timestamp	Timestamp of the creation of the activity instance
checkout_timestamp	Timestamp of the checkout of the activity instance by a stakeholder
checkin_timestamp	Timestamp of the checkin of the activity instance by a stakeholder

The *status* can be: ready-to-run, active, finished or failed.
Ready-to-run means, that the activity instance was created and is ready to be checked-out

by an activity stakeholder. Active means, that the activity was checked-out and is currently being processed. After check-in, the state changes to finished. If a failure occurs, the node's state changes to failed.

The *start parameters* must be supplied when creating an activity instance. *Return parameters* are empty until the activity was executed. The reference to the activity node that originally created it allows the system to execute a callback to the activity node that originally created the activity in case of failure or success.

The *Activity Template* is referenced in order to differentiate activity instances per Activity Template and to gain access to the data types used for start and return parameters.

Selected Staff Members lists the Staff Members, that are allowed to execute this Activity Instance per Staff Assignment Rules.

4.5.5 Process templates

Process templates stores process models by name. The permission to model or instantiate the process template is limited by abilities.

A process template thus can be expressed with the following tuple:

name	reference to process model	modeling abilities	execution abilities

name	Unique name of Process Template
reference to process model	Reference to the Process Model used in the Template
modeling abilities	Abilities required for changing the Process Template
execution abilities	Abilities required to instantiate and execute a Process Instance

The referred Process Model can be shared with Process Instances. Because the Process Model of Process Instances can be modified independently from the Process Template, the actual implementation of the application of changes to the Process Model is up to the

implementation of the Workflow Management System and not discussed to its full extend in this work.

4.5.6 Process instance

Process instances are the materialization of processes. They have a process model, they store the node's execution status and data container values.

We specify a process instance with this tuple:

reference to process template	reference to process model	data container values	node status

reference to process template	References the process template the instance is based on. If not based on one, the field is NULL
reference to process model	References the process model used
data container values	Stores the values of the data containers
node status	Stores the nodes' status information

4.6 Summary

We have presented a model for WfMS that can make use of the WfQL as interface language. This chapter was not intended as an implementation guideline nor as a complete solution for a WfMS's implementation, but rather a guide for users and implementors of the interface on how the WfMS should behave.

The prototypical implementation offers a blueprint for one possible implementation of this chapter. When existing WfMS are enhanced to offer a WfQL interface, the internal model is translated into the one presented in this chapter.

For an example implementation, see the prototypical server implementation in chapter 6.

5 Language definition

This chapter uses the previously defined WfMS realization from chapter 4, the meta models from chapter 3 and the requirements from chapter 2 to define a language for accessing WfMS. The primary intent is to show, that such a language can be formulated and to present a realization of it. We oriented ourselves as much as possible at the Structured Query Language (SQL) [SQL]. The targeted systems are quite similar, as WfMS store relational data as do Relational Database Management Systems using the SQL.

Syntax declaration
For syntax definition we are availing ourselves of the eBNF [ISO-14977]. We add use an arrow symbol ($\hookrightarrow$*reference*) to reference other entities. A reference means, that an identifier for the entity is used, opposed to the full notation of the entity. To declare one or more repetitions of a block, we use the plus sign similar to regular expressions [IEEE 1003.1]. Repetitive uses of a block are separated by comma (,). When parameters have to be inserted, they are written in *italic notation* . References and parameters are inserted via the serialization specified in sec. 5.2.4.

5.1 Modularization

In chapter 2 and 3, a number of stakeholders have been named, that use a limited scope of a WfMS's functionality. We split the WfQL's functionality on the basis of these stakeholder's probable usage patterns into modules.

Each individual module covers a specific part of the system's functionality. Some of the functionality is used by all stakeholders and is put into a base module.

We group the language parts into four categories: **Base Modules**, **Modeling Modules**, **Execution Modules** and **Analysis Modules**. Each of them has subcategories, that have specific responsibilities.

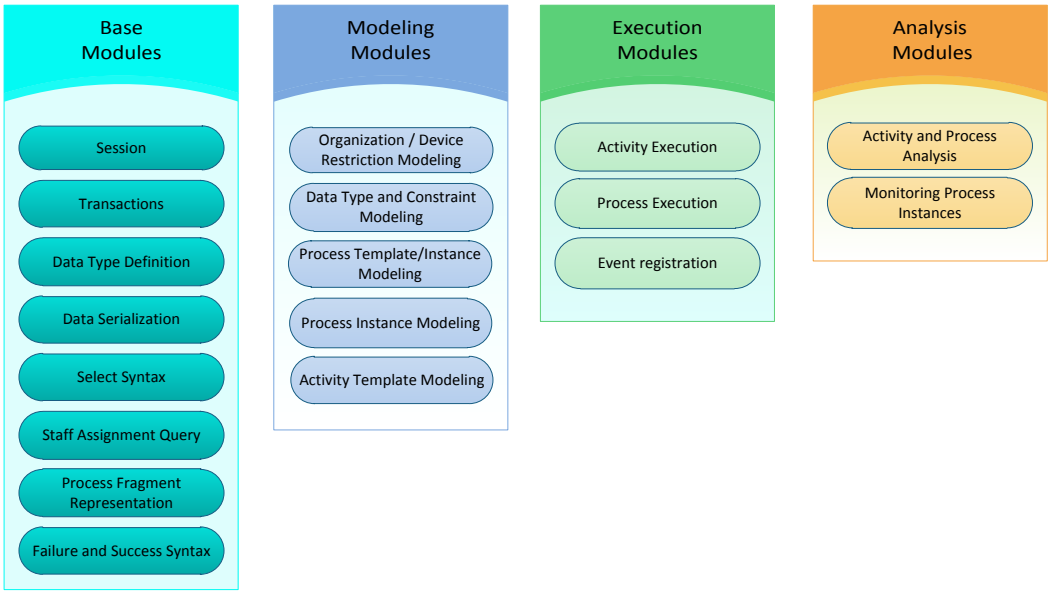


Figure 5.1: Modules Overview

Base Modules

The **Base Module** contains definitions, that form the basis of modules from the other categories.

The base modules category consists of the following modules:

Session Module	defines, how sessions are used as basis for the client's communication with the WfMS server
Transaction Module	defines, how transactions are used
Data Type Definition	defines, how data types are described
Data serialization	discusses, how the serialization is embedded
Select Syntax	defines, how attributes are selected from entities
Staff Assignment Query	defines the staff assignment syntax
Process Fragment Representation	defines a textual representation of process fragments
Failure and Success Syntax	defines a syntax that describes failures and success responses

Modeling Modules

The modeling modules contain syntax to model and change data types, process models, process templates, process instances, activity templates, the organization model and device restriction models.

The category modeling modules consist of the following modules:

Organization Model and Device Restriction Modeling	Model and modify the organization model and the device restrictions model
Data Type and Constraint Modeling	Model and modify data types and constraints on data types
Process Instance Modeling	Model and modify process instances
Process Template Modeling	Model and modify process templates
Activity Template Modeling	Model and modify activity templates

Execution Modules

The execution modules define syntax for the execution of activities and processes.

The category execution modules consist of the following modules:

Activity Execution	Check-out and check-in of activity instances
Process Execution	Trigger instantiation of processes, retrieve return parameters
Event registration and delivery	Registration of monitors and delivery of events

Analysis Modules

The analysis modules define syntax for the analysis of currently running activities, process models and process instances.

The category analysis modules consist of the following modules:

Accessing Process Instances	Allows inspection of process instances
Activity and Process Analysis	Provides analysis operations for activities and processes

5.1.1 Dependency graph

Modules in part depend on other modules. Figure 5.2 shows their dependencies. Dependencies can be used to implement a partial cover for the language. If a module and its dependencies are implemented, the support is complete. In the reverse, only the module's syntax and syntax of the depending modules may be used when implementing specific functionality.

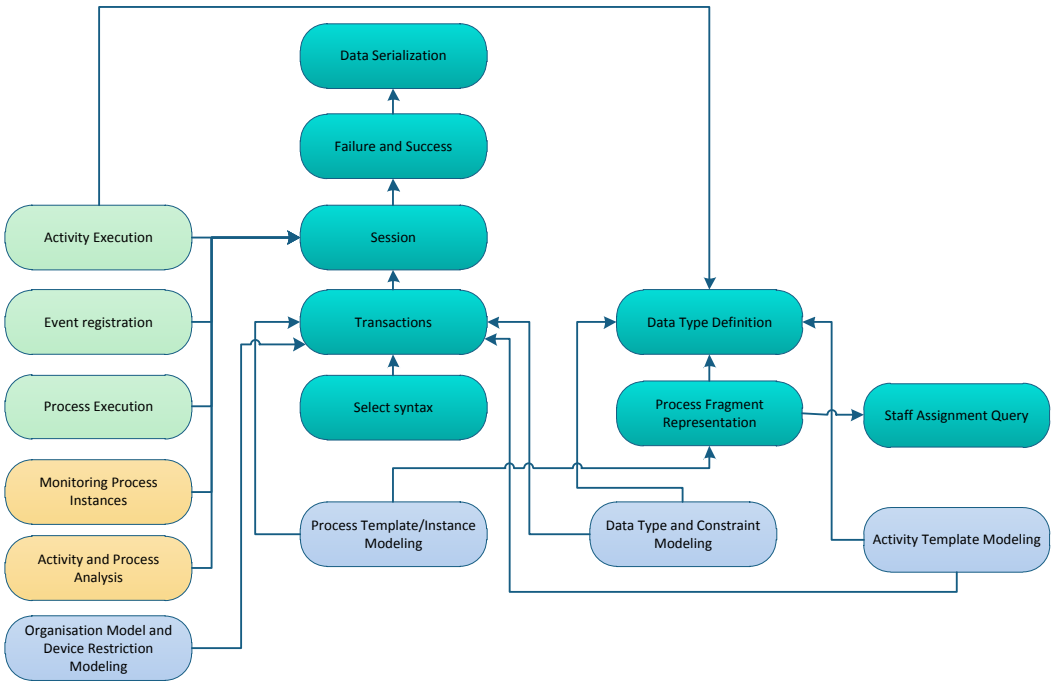


Figure 5.2: Modules Dependencies

5.2 Base modules

The category base modules defines basic syntax, that is used in multiple modules of other categories. Besides the *Select syntax* module, they offer supporting syntax that is reused in later modules. The *process fragment representation* (pfr) module for example defines syntax for representing process models that is used in modeling modules to model and modify process models.

5.2.1 Sessions

Before sending any other commands, the client hast to start a session with the WfMS. Sessions can be disconnected and reestablished. A session is started after establishing two way communication by sending the login credentials.

After a session is established, all communication is bound to the session, not the actual underlying communication method. When the underlying communication method is disconnected, due to network error or other malfunctions, the session continues to be bound to the client, and can be re-connected by the client.

<startSession> ::= "START" "SESSION" *username* *password* [*sessionID*]

Username and password are strings. The sessionID can be omitted. If present, it has to match a previously disconnected session that should get resumed.

Following a successful authentication, the server answers with this syntax:

<sessionEstablished> ::= "AOK" "SESSION" "ESTABLISHED" "WITH" "ID" *sessionID*

Possible failures:

Errorcode	Description
WRONG_USER_CREDENTIALS	User credentials could not be verified. Invalid user name or user name password combination
SESSIONID_NOT_FOUND	User credentials were matched, but the sessionID was not found or belongs to another user

If a session is not connected to a client, the communication that should be sent to the session's client is cached until the session is deleted due to timeout or a full buffer, or the session is reconnected. At no time, a session's buffer may loose data.

Disconnecting sessions

The following command issues an orderly disconnection of the session. Until the session's buffer is full or timeouts, the session can be reconnected using the start session command.

<disconnectSession> ::= "DISCONNECT" "SESSION"

The server responds with the following syntax:

<sessionDisconnected> ::= "AOK" "SESSION_DISCONNECT" "WITH" "ID" *sessionID*

Possible failures:

Errorcode	description
NO_SESSION	No session was bound, thus none can be disconnected

Terminating sessions

Terminating a session orderly disconnects and deletes it. A client may use this operation to orderly terminate its sessions, the currently used one as well as other sessions:

<terminateSession> ::= "TERMINATE" "SESSION" "WITH" "ID" *sessionID*

The server responds with the following syntax:

<sessionTerminated> ::= "AOK" "TERMINATE_SESSION" "WITH" "ID" *sessionID*

Possible failures:

Errorcode	description
NO_SESSION	No session with this ID was found, thus none can be terminated

5.2.2 Transactions

The WfQL supports non-nested transactions with *ACID* attributes. If no transaction is explicitly started, each operation is treated as if a transaction was started beforehand and committed directly after it, similar to auto commit in Database Management Systems. Transactions can, before commit, be verified. If the verification failed, the failure is stated and allows the stakeholder to fix the verification issues and try again.

Start Transaction

Transactions get started by the following syntax:

<startTransaction> ::= "START" "TRANSACTION"

Possible failures:

NESTED_TRANSACTION	A transaction is already running, no nested transactions supported

Verify Transaction

The verify command allows the client to test, whether the current transaction could be committed. If not, it replies with the errors that prohibit the commit of the transaction.

<verifyTransaction> ::= "VERIFY" "TRANSACTION"

Possible failures:

NO_TRANSACTION	No transaction running

Additional possible failures include all non-syntactical errors of the operations executed during the execution of the transaction.

Commit Transaction

The changes made to the WfMS during a transaction get applied with the commit command. If verification failures occured, the transaction is stopped and no changes get applied.

<commitTransaction> ::= "COMMIT" "TRANSACTION"

Possible failures are analogue to the `verifyTransaction` operation.

Rollback Transaction

The rollback transaction command stops the transaction without applying changes.

<rollbackTransaction> ::= "ROLLBACK" "TRANSACTION"

No possible failures.

Transaction aborted on server side

The WfMS aborts a transaction, when the data, that the transaction uses was changed. The following code is sent by the WfMS server to inform stakeholders, that their transaction was aborted:

```
<transactionAborted> ::= "ABORT" "TRANSACTION" "CAUSE" ↪Entity
```

The referenced *Entity* was changed, which was read during the execution of the transaction.

5.2.3 Data Type Definition

Data type definition defines the type of a data that gets transmitted. To define data types, we avail ourselves of the NF^2 notations [Li+88] and modify it slightly.

We predefine a number of basic data types in Table 5.1. In addition, a number of user definable data types can be expressed with the data type definition. This includes Structs, Sets, Lists, Maps and ExceptionSubTypes as shown in Table 5.2. Structs define element name type pairs, similar to C-Structs and Pascal Records. Sets and Lists aggregate elements of the same type. Maps use keys to map keys to values. ExceptionSubTypes allow declaring a payload for exceptions.

Name	Range
INTEGER	Integers
NUMBER	Floating point numbers
TEXT	String of characters
TIMESPAN	A span of time
DATE	Holds a date – second precision
BINARY	Any binary data

Table 5.1: Basic data types

Name	Range
SET<T>	Aggregates elements of the same type
LIST<T>	Similar to set, but with order
MAP<K,V>	Maps keys of type K to values of type V
STRUCT	Defines keys (string) to map to typed values
EXCEPTIONSUBTYPE	An exception data type

Table 5.2: User definable data types

<type> ::= *data_type_name* | <basicType> | <Struct> | <Set> | <List> | <Map> |
<ExceptionSubType>
<basicType> ::= "INTEGER " [":" <integerTypeConstraint>] | "NUMERIC" [":"
<numericTypeConstraint>] | "TEXT" [":" <textConstraint>] | "DATE" [":" <dateConstraint>]
| "TIMESPAN" [":" <timespanConstraint>] | "BINARY" [: <binaryConstraint>]
<structElement> ::= *elementName* ":" <type> ["," <structElement>]
<Struct> ::= "{" "STRUCT" ":" "{" <structElement> "}" "}"
<Set> ::= "{" "SET" ":" " <type> "}"
<List ::= "{" "LIST" ":" <type> "}"
<Map> ::= "{" "MAP" ":" "{" "keytype" : "{" <type> "}" "," "valuetype" : "{" <type> "}" "}" "}"
<ExceptionSubType> ::= "EXCEPTIONSUBTYPE" ":" "{" <type> "}"

Example: A type, that aggregates an identification number for an order with a customer field that is further specified in a separate type, a note that contains text and a list of products.

Given, that the customer is specified in the *Customer*-Type and the product in the *Product*-Type, we can express a struct in the data definition language:

```
"STRUCT" :{
"orderNr" : "NUMERIC",
"customer" : "Customer",
"note" : "TEXT",
"products" : { "LIST" : "Product" }
}
```

Data Type Constraints

Data type constraints constrain the values stored in data containers.

<integerTypeConstraint> ::= "{" <integerRange>+ "}"
<integerRange> ::= *minimumIntegerValue* "to" *maximumIntegerValue*

<numericTypeConstraint> ::= "{" <numericRange>+ "}"
<numericRange> ::= *minimumNumericValue* "to" *maximumNumericValue*
<textConstraint> ::= "{" ["regEx" ":" <regularExpression>","] ["text_length" ":"
<integerTypeConstraint>"]}"
<dateConstraint> ::= "{" ["hour" : *hour* ","] ["minute" : *minute*] ["dayOfTheWeek" ":"
dayOfTheWeek ","] ["dayOfTheMonth" ":" *dayOfTheMonth* ","] ["monthOfTheYear" ":"
monthOfTheYear ","] ["year" ":" *year* ","] ["startDate" ":" *startDate* ","] ["endDate"
":" *endDate*] "}"
<timespanConstraint> ::= "{" ["minimumLength" ":" *minimumLengthInSeconds* ","] [
"maximumLength" ":" *maximumLengthInSeconds* "] }"
<binaryConstraint> ::= "{" ["maxSize" ":" *maxSize* ","] ["minSize" ":" *minSize*] "}"

minimumIntegerValue	Minimum Integer Value (whole number)
maximumIntegerValue	Maximum Integer Value (whole number)
minimumNumericValue	Minimum Numeric Value (decimal number)
maximumNumericValue	Maximum Numeric Value (decimal number)
hour	Allowed hour (0-23)
minute	Allowed minute (0-59)
dayOfTheWeek	Day of the week (0-6)
dayOfTheMonth	Day of the month (0-31)
monthOfTheYear	Day of the Year (1-366)
year	Year (whole number)
startDate	Minimum date (date type)
endDate	Maximum date (date type)
minimumLengthInSeconds	Minimum length in seconds (whole number)
maximumLengthInSeconds	Maximum length in seconds (whole number)
maxSize	Maximum size in bytes (whole number)
minSize	Minimum size in bytes (whole number)
regEx	A regular expression, that is required to be matched for the text data to be accepted.

5.2.4 Data serialization

Any data sent via the WfQL has a type. Therefore, we can send serialized data without the type definition, because it is either known to the recipient or the recipient can request the data type definition. A number of data representation protocols exist, for example XML[ISO 8879], JSON[JSON] and the Lisp notation[LISP].

For data serialization we require a protocol that has the following properties:

1. Fast computerized parsing
2. Low data overhead
3. Subjectively easy to read for humans
4. Trivial to write parsers
5. Embeddable into the language

We are not specifying which data transport protocol should be used. In chapter 6 we will make a design decision and argue our point. For the sake of the language specification, no specific serialization method is selected and it is open to the implementor to decide, which one is suited best.

A discussion about data serialization is held in in sec. 7.2.

5.2.5 Select syntax

In chapter 4, entities were introduced with their attributes. The language needs a way to access these attributes.

Introducing select and where SQL allows access to attributes – or rather, columns – of tables as selecting a subset of the columns of a table's rows. If all are selected, each row is treated as a tuple. The WfQL uses the same paradigm.

Example syntax to access the start parameters of all activity templates:

```
SELECT start_parameter FROM activity_templates
```

SQL allows for filtering of the data set before selecting, which is expressed by the WHERE clause. It formulates a boolean expression, that is evaluated per row with access to its the attributes, similar to the SQL.

Example syntax to access the start parameters of all activity templates with the name “call customer” and “call shipping department”:

```
SELECT start_parameters, name FROM activity_templates
WHERE name = 'call customer' OR name = 'call shipping department'
```

In order to make the language more flexible, sub-queries are allowed. That means, that any query can contain any other query and use its return value in the place of the sub-query.

```
SELECT name FROM (
SELECT activity_template FROM activity_instance WHERE state = failed) )
```

The sub-query, or sometimes called nested query, is evaluated first. It evaluates to a set of activity template that have activity instances that have failed. From this set, the name is selected.

Thus, it is possible, to access any attribute of any entity and use it.

Formal definition of select syntax:
We formulate the definition of the select .. from .. where syntax analogue to the definition in the SQL:

```
<entitySet> ::= "ACTIVITY_TEMPLATES" | "ACTIVITY_INSTANCES" |
"PROCESS_TEMPLATES" | "PROCESS_INSTANCES" | "DATA_TYPES" |
"ORGGROUP" | "ORGUNITS" | "PROJECTTEAM" | "POSITION" | "ROLE" | "ABILITY"

<fromDefinition> ::= ["SELECT"  attributes +] "FROM" <entitySet> ["WHERE"  <term> ]
```

`SELECT` can be omitted. When it is not omitted, the attributes define the attributes of the EntitySet, that should be in the result tuple of the operation. Attributes is a comma separated list of the attribute’s names.

`WHERE` can be omitted. When not omitted, its `term` is evaluated per entity of the EntitySet. When the term is evaluated to true, the entity is added to the result set.

The server response is a ordered list of tuples. Expressed in the Data Type Definition Language:

```
List { Struct { (attributeName : attributeType)+ } }
```

The response heavily relies on the used data serialization discussed in sec. 5.2.4. With the entity's definition available, the response can be parsed correctly and used in the client.

```
<term> ::= [<negateOperator>] <term> | <value> <compareSymbol> <value> | <term>
<logicalOperator> <term>
<compareSymbol> ::= "==" | "! =" | ">" | "<" | "<=" | ">="
<logicalOperator> ::= "&&" | "||"
<negateOperator> ::= "!"
<value> ::= text value | numeric value | date value | timespan value | attribute
```

When using *attribute* in a value, it references an attribute of elements of the *EntitySet* .

5.2.6 Staff Assignment Query

The Staff Assignment Query (SAQ) uses the `term` in order to select a part of the organization model (or the entire) which is ultimately referencing *Staff Members*. It is used in *Activities (in Process Models)* and Process Fragments in general (which subordinated Process Fragments inherit the SAQ) and expresses, which Staff Members may execute a consequently created *Activity Instance*. The SAQ allows access to data containers of process instances. With this functionality in mind, it is possible to assign Staff Members based on the Organization Model and the already executed part of the Process Instance and to, for example, restrict the execution of an activity if a certain (or any) other activity of the process instance was executed by the Staff Member, Organization Unit, Project Team et cetera before.

The following definition ultimately selects Staff Members by use of the Organization Model:

```
<saTerm> ::= ( "(" <saTerm> ")" ) | ( <saTerm> "IN" <saTerm> ) | ( <negateOperator>
<saTerm>) | ( <organizationModelEntity> <compareSymbol> <value> ) | ( <saTerm>
<logicalOperator> <saTerm> ) | <hierarchySaTerm>
<hierarchySaTerm> ::= ( <organizationModelEntity> <compareSymbol> <WFQL> ) | (
<hierarchicalOrModelEntity> [TRANSITIVE] "SUPERORDINATED"|"SUBORDINATED"
<hierarchicalOrModelEntity> [<compareSymbol> <saTerm>|<value>] )
<projectTeamOrgModelEntity ::= "PROJECT_TEAM" ["." "leadBy" ]
<orgUnitOrgModelEntity> ::= "ORG_UNIT" ["." "leadBy" ]
<hierarchicalOrgModelEntity> ::= <projectTeamOrgModelEntity> |
"<orgUnitOrgModelEntity> | "ROLE"
<organizationModelEntity> ::= <hierarchicalOrgModelEntity> |
<orgGroupOrgModelEntity> | "POSITION" | "STAFF_MEMBER" | "ABILITY"
<orgGroupOrgModelEntity> ::= "ORG_GROUP" [ "." "leadBy" ]
```

Non-hierarchical Entity relationships are accessible through the SAQ via the following syntax:

Entity	Relation	Description
ORG_UNIT	leadBy	Returns the Position that leads the organization unit
ORG_GROUP	leadBy	Returns the Position that leads the organization group
PROJECT_TEAM	leadBy	Returns the Position that leads the project team

To select any `organizationModelEntity`, the entity type is compared to a unique identifier. For example:

```
PROJECT_TEAM == "Team Alpha"
```

This would select the project team with the name "Team Alpha" from the organization model. At the point of use, the project team entity is used. Staff assignment however requires Staff Members. Therefore, all staff members that stand in relation to project team "Team Alpha" would be selected. The search for all related staff members is however executed as last as possible, so the project team entity is available for further operations. The project team has the "leads" relationship to positions and which can be used to query for the leading position via the `projectTeamOrgModelEntity` definition. In addition, all operations are applied on all entities of an entity set. The results are added to the result set of the operation. It is therefore possible to query for the leading positions of multiple project

teams (organization groups and organization units analogue) by applying the `leadBy` operator on a set of entities, for example:

```
( PROJECT_TEAM == "Team Alpha" || PROJECT_TEAM == "Team Beta" ).leadBy
```

Furthermore, the `WFQL` operator `IN` allows for check, if an entity is part of a set of entities. Therefore, it is possible to select positions and check, whether they lead a set of project teams. For example:

```
( POSITION == "Senior Developer" ) IN (PROJECT_TEAM == "Team Alpha" ||
PROJECT_TEAM == "Team Beta" ).leadBy
```

The query would result an empty set or the position of "Senior Developer" if it was leading project team "Team Alpha" or "Team Beta". Ultimately, based on the position, staff members would be selected.

In addition, the SAQ allows for use of the entity set `THIS_PROCESS_INSTANCE` that is referencing the Process Instance, that the Staff Assignment Query is executed in.

All organization model entities in `hierarchicalOrgModelEntity` can form a recursive hierarchy. For example: Given Project Team P_1 has a subordinated Project Team P_2 that has a subordinated Project Team P_3. P_2 is a directly subordinated of P_1. P_3 is not directly subordinated of P_1. P_2 and P_3 are transitively subordinated of P_1. The `hierarchySaTerm` defines, how hierarchical structures are tested for their structure. Given, we would like to select all staff members that have the position "SOFTWARE ARCHITECT", the Role "DEVELOPER" in a Project Team, that is directly subordinated to P_1, we could formulate the following query:

```
POSITION == "SOFTWARE ARCHITECT" && ROLE == "DEVELOPER" &&
PROJECT_TEAM SUBORDINATED PROJECT_TEAM == "P_1"
```

Analogue for ORG_UNIT and ROLE.

With the SAQ, we created a syntax to make full use of the organization model's power in a SQL-a-like syntax.

5.2.7 Process Fragment Representation

The Process Fragment Representation represents fragments of a process's model. The representation heavily relies on the Adept [Re00] Meta-Model modifications: Exception blocks, Data typification with constraints,block-local data containers and dynamically parallel blocks.

Process Fragment Representation (Pfr) can expresses activities and blocks:

```
<pfr> := <activity>|<block>
```

Activities are defined with a unique name. Activity start parameters are mapped from data containers to start parameters via the read declaration, return parameters analogue with write. The activity template is referenced with its unique name.

```
<getter> := ".GET" "(" elementPosition | key ")"
<setter> := ".PUT" "(" key ")"
<mapping> := from [<getter>] "TO"|"APPEND" to [<setter>][,<mapping>]
<staffAssignmentRules> := "SAR" ":" <saTerm>
<read> := "READ" "(" <mapping> ")"
<write> := "WRITE" "(" <mapping> ")"
<synchronize> ::= "SYNCHRONIZE" "WITH" ↪ProcessFragmentName
<activity> := "ACTIVITY" activityName activityTemplate "("
   [waiting_timeout = timeout in milliseconds ],
   [execution_timeout = timeout in milliseconds ],
   [<synchronize>+],
   [<read>+,]
   [<write>+,]
   [<staffAssignmentRules>+]
   [<exceptionBlock>+]
   ")"
```

The mapping in read and write operations offer access to data containers. For reading, the `TO` construct is used. For writing, either the `TO` construct which overwrites any possibly existing value. Alternatively, `APPEND` can be used for writing, which expresses appending

data to aggregating data types `Set` and `Lists`. Appending is only possible, if the source of the mapping reflects the type of the Set or List.

Getters are use for `Maps` and Lists. The elementPosition expresses the position of an element in a list, the key the key of a Map. `PUT` is used to put elements into a Map.

The `synchronize` syntax expresses synchronization edges to other process fragments.

Blocks are used to design the control flow of process models. They can define exception catch blocks that handle failures and define data containers, whose name-space is limited to the block and its inner process fragments.

All types of blocks have a block headers with data definitions. They define data containers, that are block specific. In addition, it is possible to `READ` and `WRITE` data containers from the outer name-space. In this respect, we differ from the Adept model and allow a hierarchical name-space with local data containers.

`ExceptionBlock`s differ from normal blocks in so far, that they map the exception that occurred to a data container, and that their inner process fragment is terminated by either an exception jump or an exception throw.

<block> := <cfblock>,<serialblock>
<serialBlock> := "SERIALBLOCK *blockName* "(" <blockCommons>, <pfr>+,
<exceptionBlock>+ ")"

<blockCommons> :=
[<dataDefinition>,][<read>,][<write>,][<staffAssignmentRules>,][<synchronize>,]
<dataDefinition> := "DATA" <type> *dataContainerName* [,<dataDefinition>]

<exceptionBlock> := "EXCEPTION *exceptionName* "(" <blockCommons> "WRITE "("
"EXCEPTION" "TO" *dataContainer* ")", <exceptionPfr> ")"
<exceptionPfr> := [<pfr>,]<exceptionXor>|<exceptionJump>|<exceptionThrow>
<exceptionXor> := EXCEPTION_XOR *exceptionXorName* (<blockCommons>, (
"CASE" <decisionPredicat> ":" <exceptionPfr>)+)
<exceptionJump> := EXCEPTIONJUMP *jumpName* TO ↪*pfr*

A process model can be described with this syntax, that is block structured and makes use of exceptions. Exception blocks are limiting the use of exception jumps and exception

throws to be either at the end of the exception block, or the last node of a exceptionXor. The latter is in essence a normal xor, but is specifically only allowed as the last Process Fragment of a exception block to make sure, only one exception jump and only one exception throw respectively can be modeled.

Control Flow Blocks

Control flow blocks define process fragments, that are used to model the process model's control flow. We orient the naming and the behavior at the Adept meta model.

XOR Blocks are defined with the following syntax:

<xorBlock> := "XOR" *xorBlockName* "("
<blocksCommons>,
("CASE" <decisionPredicat> ":" "(" <pfr> ")")+,
<exceptionBlock>+
")"
")"

AND Blocks are slightly different from XOR Blocks. They do not require a decision logic, but can be joined by different constructs, such as XOR, AND and OR.

<andBlock> := "AND" ("AND"|"OR"|"XOR") *xorBlockName* "("
<blocksCommons>,
("BRANCH" "(" <pfr> ")")+,
<exceptionBlock>+
")"
")"

OR Blocks combine the attributes of XOR Blocks and AND Blocks. They require a decision logic, and can be joined in different ways:

```
<orBlock> := "OR" ("AND"|"OR"|"XOR")  orBlockName  "("
<blocksCommons>,
("CASE" <decisionPredicat> ":" "(" <pfr> ")" )+,
<exceptionBlock>+
")"
")"
```

Loop blocks are executed n-times, as long as a decision predicate evaluates to true:

```
<loopBlock> := "LOOP"  orBlockName  "("
<blocksCommons>,
"WHILE" <decisionPredicat> ":" "(" <pfr> ")",
<exceptionBlock>+
")"
")"
```

Dynamic parallel blocks express dynamical parallelism. They require a data container of the type set or list as input:

```
<dynParBlock> := "DYNPAR"  dynParName  "("
<blocksCommons>,
"READ" "("  dataContainer  "TO"  localDataContainer  ")",
<pfr>,
<exceptionBlock>+
")"
")"
```

dataContainer	required to be a set or list data container
localDataContaier	required to be defined in `<blocksCommons>` and of the type of an element of the *dataContainer*.

In summary, control flow blocks are defined per:

```
<cfBlock> := <dynParBlock> | <loopBlock> | <xorBlock> | <orBlock> | <andBlock>
```

Process Fragment Representation Examples

To illustrate the use of the Pfr, we show its use in the form of the following examples.

Example: Simple order-to-delivery process
This example stems from the introduction chapter. Figure 1.1 shows a simple order-to-delivery process.

The process serially executes three activities: A stakeholder in customer service, that records the order of products. A stakeholder in book-keeping, that executes an invoice and a shipping stakeholder, that ships the ordered products. Customer service records the ordered products, shipping address, invoice number and total price. Invoice reads the total price and invoicenumber. Shipping reads the ordered products and shipping address.

To picture this process in the Pfr, we use the SerialBlock construct, that executes its inner Pfrs serially. The data flow is realized via data containers, that are declared in the SerialBlock. Let us assume, that activity templates exist, and that they match the mapping expressed in the following example:

```
SERIALBLOCK OrderToDeliver (
  DATA orderedProducts LIST : Products,
  DATA shippingAddress Address,
  DATA invoiceNumber INTEGER,
  DATA totalPrice NUMBER,
  ACTIVITY RecordOrderOfProducts RecordOrders
  (
    WRITE ( products TO orderedProducts ),
    WRITE ( address TO shippingAddress ),
    WRITE ( generatedNumber TO invoiceNumber ),
    WRITE ( calculatedPrice TO totalPrice )
  ),
  ACTIVITY Invoice InvoiceMoney
  (
    READ ( totalPrice TO amount ),
    READ ( invoiceNumber TO number ),
  ),
  ACTIVITY ShippingOfProducts ShipProducts
  (
    READ ( orderedProducts TO products ),
    READ ( shippingAddress TO address )
  )
)
```

5.2.8 Failure and success notifications

The WfQL offers a set of failure classes, that are used to express problems that occurred while trying to execute a command. Syntax Errors are caused by malformed syntax and are thus reported with the parsing error that occurred, at the position of the failure:

```
<syntaxError> ::= "SYNTAX" "ERROR" "FOUND" "NEAR"  position  "(" ( "EXPECTED"
"'" expected_character "'" ) | ( "CHARACTER" "NOT" "ALLOWED" "'"
disallowed_character "'" ) | ( "COMMAND" "UNKNOWN"  command_name  ) ")"
```

The Type Error is returned, when a value of a specific type was expected, but another was received:

<typeError> ::= "TYPE" "ERROR" "FOUND" "NEAR" *position* "," \" *nearby_syntax* \" "EXPECTED" "TYPE" <type>

Permission denied error is returned, when a stakeholder tries executing an operation, that he or she has not the permission level to do so:

<permissionDeniedError> ::= "PERMISSION" "DENIED" "FOR" *operation* "NEAR" *position* , \" *nearby_syntax* \"

Command specific errors name the reason for an error that is defined per operation that it might occur:

<commandSpecificError> ::= "COMMAND" "ERROR" *errorReason*

Success notifications

When a command was executed successfully, the server responds with a simple, serialized Text: "AOK".

5.3 Modeling Modules

All syntax that is used to create and modify models is contained in the category modeling modules. The organization model and device restriction modeling module governs the creation and modification of the organization model and the device restriction model. The data type and constraint modeling module covers the creation and modification of user defined data types with constraints. The module process template and process instance modeling covers the creation and modification of process templates and the modification of process models of process instances. The process instance modeling module covers additional process instance specific modeling operations. The activity template modeling module covers the creation and modification of activity templates.

5.3.1 Organization Model and Device Restriction Modeling

This module covers creating and modifying the organization model's structure and the device restriction model. The latter is rather simple in comparison to the organization model and orients itself at the syntax of the organization model.

Organization Model

The organization model was presented in Figure 4.2 and specified in sec. 4.2. The following paragraphs show, how entities of the organization model are created and modified. Relations are numbered with roman numbers according to the specifications in sec. 4.2.

Organization Group

The following syntax creates an organization group with the given name, and an optional lead position:

<createOrgGroup> ::= "CREATE" "ORG_GROUP" "WITH" "name" "=" *name* ["," "lead_position" "=" ↪*POSITION*]

Parameters:

name	Name of the Organization Group, unique
lead_position	Reference to position that leads the organization group

Possible failures:

UNIQUE_NAME_CONSTAINT	Organization group with this name already exists
POSITION_NOT_FOUND	The supplied position could not be found

Alter syntax to set a new name and lead position respectively:

<alterOrgGroupName> ::= "ALTER" "ORG_GROUP" "SET" "name" "=" *name* ["WHERE" <term>]
<alterOrgGroupLeadPos> ::= "ALTER" "ORG_GROUP" "SET" "lead_position" "=" ↪*POSITION* ["WHERE" <term>]

Failures analogue to create.

Organization Unit

The following syntax creates an organization unit with name, optional lead position and optional superordinate organization unit:

<createOrgUnit> ::= "CREATE" "ORG_UNIT" "WITH" "name" "=" *name* ["," "lead_position" "=" ↪*POSITION*] ["," "higher_org_unit" "=" ↪*ORG_UNIT*]

Parameters:

name	Name of the organization unit, unique
lead_position	Reference to position that leads the organization unit
higher_org_unit	Reference to superordinate organization unit

Possible failures:

UNIQUE_NAME_CONSTAINT	Organization group with this name already exists
POSITION_NOT_FOUND	The supplied position could not be found
ORG_UNIT_NOT_FOUND	The supplied organization unit could not be found

Alter syntax to set a new name, lead position and superordinate organization unit respectively:

<alterOrgUnitName> ::= "ALTER" "ORG_UNIT" "SET" "name" "=" *name* ["WHERE" <term>]
<alterOrgUnitLeadPos> ::= "ALTER" "ORG_UNIT" "SET" "lead_position" "=" ↪*POSITION* ["WHERE" <term>]
<alterOrgUnitHigherOrgUnit> ::= "ALTER" "ORG_UNIT" "SET" "higher_org_unit" "=" ↪*ORG_UNIT* ["WHERE" <term>]

Failures analogue to create.

Project Team

The following syntax creates a project team with name, optional lead position and optional superordinate project team:

<createProjectTeam> ::= "CREATE" "PROJECT_TEAM" "WITH" "name" "=" *name* "," "lead_position" "=" ↪*POSITION* ["," "higher_project_team" "=" ↪*PROJECT_TEAM*

name	Name of project team (unique)
lead_position	Reference to position that leads the project team
higher_project_team	Reference to superordinate project team

Possible failures:

UNIQUE_NAME_CONSTAINT	A Project Team with this name already exists
POSITION_NOT_FOUND	The supplied position could not be found

Alter syntax to set a new name, lead position and superordinate project team respectively:

<alterProjectTeamName> ::= "ALTER" "PROJECT_TEAM" "SET" "name" "=" *name* ["WHERE" <term>]
<alterProjectTeamLeadPos> ::= "ALTER" "PROJECT_TEAM" "SET" "lead_position" "=" ↪*POSITION* ["WHERE" <term>]
<alterProjectTeamtHigherProjectTeam> ::= "ALTER" "PROJECT_TEAM" "SET" "higher_project_team" "=" ↪*PROJECT_TEAM* ["WHERE" <term>]

Failures analogue to create.

Position

The following syntax creates a position with name and organization unit it belonging to:

<createPosition> ::= "CREATE" "PROJECT" "WITH" "name" "=" *name* "," "org_unit" "=" ↪*ORG_UNIT*

name	Name of project (unique)
org_unit	Reference to Organization Unit this Position is participating in

Possible failures:

UNIQUE_NAME_CONSTAINT	A Project with this name already exists
ORG_UNIT_NOT_FOUND	The supplied Organization Unit could not be found

Alter syntax to set a new name and organization unit respectively:

<alterPositionName> ::= "ALTER" "POSITION" "SET" "name" "=" *name* ["WHERE" <term>]
<alterPositionOrgUnit> ::= "ALTER" "POSITION" "SET" "org_unit" "=" *org_unit* ["WHERE" <term>]

Failures analogue to create.

Role

The following syntax creates a role with name and optional superordinate role:

<createRole> ::= "CREATE" "ROLE" "WITH" "name" "=" *name* ["," "higher_role" "=" ↪*ROLE*]

name	Name of Role (unique)
higher_role	Reference to hierarchically higher Role

Possible failures:

UNIQUE_NAME_CONSTAINT	A Role with this name already exists
ROLE_NOT_FOUND	The supplied Role could not be found

Alter syntax to set a new name and superordinate role respectively:

<alterRoleName> ::= "ALTER" "ROLE" "SET" "name" "=" *name* ["WHERE" <term>]
<alterRoleHigherRole> ::= "ALTER" "ROLE" "SET" "higher_role" "=" ↪*ROLE* ["WHERE" <term>]

Failures analogue to create.

Ability

The following syntax creates an ability with the given name:

<createAbility> ::= "CREATE" "ABILITY" "WITH" "name" "=" *name*

name	Name of Ability (unique)

Possible failures:

UNIQUE_NAME_CONSTAINT	A Role with this name already exists

Alter syntax to change the name of an ability:

<alterRole> ::= "ALTER" "ABILITY" "SET" "name" "=" *name* ["WHERE" <term>]

Failures analogue to create.

Staff Member

The following syntax creates a new staff member with a name:

<createStaffMember> ::= "CREATE" "STAFF_MEMBER" "WITH" "name" "=" *name*	
name	Name of Staff Member (unique)

Possible failures:

UNIQUE_NAME_CONSTAINT	A Staff Member with this name already exists

Alter syntax to set a new name:

<alterStaffMember> ::= "ALTER" "STAFF_MEMBER" "SET" "name" "=" *name* ["WHERE" <term>]

Failures analogue to create.

Substitution Rule

The following syntax creates a substitution rule with name, optional start date and optional end date:

<createSubstitutionRule> ::= "CREATE" "SUBSTITUTION_RULE" "WITH" "name" "=" *name* ["," "start_date" "=" *start_date*] ["," "end_date "=" *end_date*]

Parameters:

name	Name of the Substitution Rule (unique)
start_date	Starting date of the substitution rule (optional)
end_date	Ending date of the substitution rule (optional)

Possible failures:

UNIQUE_NAME_CONSTRAINT	A Substitution Rule with this name already exists
DATE_CONSTRAINT	End date is before Start date

Alter syntax to change the substitution rule's name, start date and end date respectively:

<alterSubstitutionRuleName> ::= "ALTER" "SUBSTITUTION_RULE" "SET" "name" "=" *name* ["WHERE" <term>]
<alterSubstitutionRuleStartDate> ::= "ALTER" "SUBSTITUTION_RULE" "SET" "start_date" = *start_date* ["WHERE" <term>]
<alterSubstitutionRuleEndDate> ::= "ALTER" "SUBSTITUTION_RULE" "SET" "end_date" "=" *end_date* ["WHERE" <term>]

Failures analogue to create.

Modeling organization model relationships

The relationships presented in sec. 4.2 are modeled with the following syntax definitions. The numbering is analogue.

Relationship I

Relationship I is a many to many relationship that models the "forms" relationship of organization units with organization groups.

<alterOrgGroupOrgUnitAdd> ::= "ALTER" "ORG_GROUP_ORG_UNIT" "ADD" "(" ↪*ORG_GROUP* "," ↪*ORG_UNIT* ")"
<alterOrgGroupOrgUnitRem> ::= "ALTER" "ORG_GROUP_ORG_UNIT" "REMOVE" "(" ↪*ORG_GROUP* "," ↪*ORG_UNIT* ")"

Possible failures:

ORG_GROUP_NOT_FOUND	The supplied organization group could not be found
ORG_UNIT_NOT_FOUND	The supplied organization unit could not be found

Relationship II

Relationship II is a many to many relationship that models the "consists of" relationship of project teams with organization units.

<alterProjectTeamOrgUnitAdd> ::= "ALTER" "PROJECT_TEAM_ORG_UNIT" "ADD" "("
↪*PROJECT_TEAM* "," ↪*ORG_UNIT* ")"
<alterProjectTeamOrgUnitRem> ::= "ALTER" "PROJECT_TEAM_ORG_UNIT"
"REMOVE" "(" ↪*PROJECT_TEAM* "," ↪*ORG_UNIT* ")"

Possible failures:

ORG_GROUP_NOT_FOUND	The supplied organization group could not be found
PROJECT_TEAM_NOT_FOUND	The supplied project team could not be found

Relationship III

Relationship III is a many to many relationship that models the “consists of” relationship of project teams with positions.

<alterProjectTeamPositionAdd> ::= "ALTER" "PROJECT_TEAM_POSITION" "ADD" "("
↪*PROJECT_TEAM* "," ↪*POSITION* ")"
<alterProjectTeamPositionRem> ::= "ALTER" "PROJECT_TEAM_POSITION"
"REMOVE" "(" ↪*PROJECT_TEAM* "," ↪*POSITION* ")"

Possible failures:

POSITION_NOT_FOUND	The supplied Position could not be found
PROJECT_TEAM_NOT_FOUND	The supplied Project Team could not be found

Relationship IV

Relationship IV is a many to many relationship that models the “staffs” relationship of staff members with positions.

<alterStaffMemberPositionAdd> ::= "ALTER" "STAFF_MEMBER_POSITION" "ADD" "("
↪*STAFF_MEMBER* "," ↪*POSITION* ")"
<alterStaffMemberPositionRem> ::= "ALTER" "STAFF_MEMBER_POSITION"
"REMOVE" "(" ↪*STAFF_MEMBER* "," ↪*POSITION* ")"

Possible failures:

POSITION_NOT_FOUND	The supplied Position could not be found
STAFF_MEMBER_NOT_FOUND	The supplied Staff Member could not be found

Relationship V

Relationship V is a many to many relationship that models the “describes” relationship of position with roles.

```
<alterRolePositionAdd> ::= "ALTER" "ROLE_POSITION" "ADD" "(" ↪ROLE "," ↪POSITION ")"
<alterRolePositionRemove> ::= "ALTER" "ROLE_POSITION" "REMOVE" "(" ↪ROLE "," ↪POSITION ")"
```

Possible failures:

POSITION_NOT_FOUND	The supplied Position could not be found
ROLE_NOT_FOUND	The supplied Role could not be found

Relationship VI

Relationship VI is a many to many relationship that models the “substitution” relationship of roles with substitution rules.

```
<alterSubstitutionRuleRoleAdd> ::= "ALTER" "SUBSTITUTION_RULE_ROLE" "ADD" "(" ↪SUBSTITUTION_RULE "," ↪ROLE ")"
<alterSubstitutionRuleRoleRem> ::= "ALTER" "SUBSTITUTION_RULE_ROLE" "REMOVE" "(" ↪SUBSTITUTION_RULE "," ↪ROLE ")"
```

Possible failures:

SUBSTITUTION_RULE_NOT_FOUND	The supplied Substitution Rule could not be found
ROLE_NOT_FOUND	The supplied Role could not be found

Relationship VII

Relationship VII is a many to many relationship that models the “substitution” relationship of positions with substitution rules.

<alterSubstitutionRulePositionAdd> ::= "ALTER" "SUBSTITUTION_RULE_POSITION" "ADD" "(" ↪*SUBSTITUTION_RULE* "," ↪*POSITION* ")"
<alterSubstutitioNRulePositionRem> ::= "ALTER" "SUBSTITUTION_RULE_POSITION" "REMOVE" "(" ↪*SUBSTITUTION_RULE* "," ↪*POSITION* ")"

Possible failures:

SUBSTITUTION_RULE_NOT_FOUND	The supplied Substitution Rule could not be found
POSITION_NOT_FOUND	The supplied Position could not be found

Relationship VIII

Relationship VIII is a many to many relationship that models the “has” relationship of roles with abilities.

<alterRoleAbilityAdd> ::= "ALTER" "ROLE_ABILITY" "ADD" "(" ↪*ROLE* "," ↪*ABILITY* ")"
<alterRoleAbilityRem> ::= "ALTER" "ROLE_ABILITY" "REMOVE" "(" ↪*ROLE* "," ↪*ABILITY* ")"

Possible failures:

ABILITY_NOT_FOUND	The supplied Ability could not be found
ROLE_NOT_FOUND	The supplied Role could not be found

Relationship IX

Relationship IX is a many to many relationship that models the “has” relationship of staff members with abilities.

<alterStaffMemberAbilityAdd> ::= "ALTER" "STAFF_MEMBER_ABILITY" "ADD" "(" ↪*STAFF_MEMBER* "," ↪*ABILITY* ")"
<alterStaffMemberAbilityRem> ::= "ALTER" "STAFF_MEMBER_ABILITY" "REMOVE" "(" ↪*STAFF_MEMBER* "," ↪*ABILITY* ")"

Possible failures:

STAFF_MEMBER_NOT_FOUND	The supplied Staff Member could not be found
ROLE_NOT_FOUND	The supplied Role could not be found

Deletion of entities

The following syntax is used for any delete operation on the organization model:

<deleteFromOrgModel> ::= "DELETE" "FROM" (<organizationModelEntity> |
"SUBSITUTION_RULE") "WHERE" <term>

Device Restriction Modeling

The device restriction modeling creates and modifies the device restrictions presented in sec. 4.2.3. It is an ad hoc model, that refers the ability from the organization model. It depends on the organization model, the organization model does not depend on the device restriction model.

Restriction

The following syntax creates a new restriction with a name and a reference to an ability that the restriction restricts.

<createRestriction> ::= "CREATE" "RESTRICTION" "WITH" "name" "=" *name* ","
"ability" "=" ↪*ABILITY*

Parameters:

name	Restriction name, unique
ability	Reference to ability

Possible failures:

UNIQUE_NAME_CONSTRAINT	A restriction with this name already exists
ABILITY_NOT_FOUND	Referenced ability was not found

The following alter syntax modifies a restriction and sets a new name and a new ability respectively:

<alterRestrictionName> ::= "ALTER" "RESTRICTION" "SET" "name" "=" *name*
["WHERE" <term>]
<alterRestrictionAbility> ::= "ALTER" "RESTRICTION" "SET" "ability" "=" ↪*ABILITY*
["WHERE" <term>]

Failures analogue to create.

Device class

The following syntax creates a new device class with a name:

<createDeviceClass> ::= "CREATE" "DEVICE_CLASS" "WITH" "name" "=" *name*

Parameters:

name	Device class name (unique)

Possible failures:

UNIQUE_NAME_CONSTRAINT	A Device Class with this name already exists

The following alter syntax modifies the name of a device class:

<alterDeviceClassName> ::= "ALTER" "DEVICE_CLASS" "SET" "name" "=" *name*

Failures analogue to create.

Specific Device The following create syntax creates a new specific device with a name:

<createSpecificDevice> ::= "CREATE" "SPECIFIC_DEVICE" "WITH" "name" "=" *name*

Parameters:

name	Specific device's name (unique)

Possible failures:

UNIQUE_NAME_CONSTRAINT	A Specific Device with this name already exists

The following alter syntax modifies a specific device and sets a new name:

<alterSpecificDeviceName> ::= "ALTER" "SPECIFIC_DEVICE" "SET" "name" "=" *name* ["WHERE" <term>]

Failures analogue to `createSpecificDevice`.

Device restriction relationships

The following relationships are defined in sec. 4.2.3. The numbering is analogue.

Relationship I

Relationship I is a many to many relationship that forms the “restricts” relationship of device classes with restrictions.

<alterDeviceClassRestrictionAdd> ::= "ALTER" "DEVICE_CLASS_RESTRICTION"
"ADD" "(" ↪*DEVICE_CLASS* "," ↪*RESTRICTION* ")"
<alterDeviceClassRestrictionRem> ::= "ALTER" "DEVICE_CLASS_RESTRICTION"
"REMOVE" "(" ↪*DEVICE_CLASS* "," ↪*RESTRICTION* ")"

Possible failures:

DEVICE_CLASS_NOT_FOUND	Referenced Device class not found
RESTRICTION_NOT_FOUND	Referenced Restriction not found

Relationship II

Relationship II is a many to many relationship that forms the “belongs” relationship of specific devices with device classes.

<alterDeviceClassSpecificDeviceAdd> ::= "ALTER"
"DEVICE_CLASS_SPECIFIC_DEVICE" "ADD" "(" ↪*DEVICE_CLASS* ","
↪*SPECIFIC_DEVICE* ")"
<alterDeviceClassSpecificDeviceRem> ::= "ALTER"
"DEVICE_CLASS_SPECIFIC_DEVICE" "REMOVE" "(" ↪*DEVICE_CLASS* ","
↪*SPECIFIC_DEVICE* ")"

Possible failures:

DEVICE_CLASS_NOT_FOUND	Referenced Device class not found
SPECIFIC_DEVICE_NOT_FOUND	Referenced Specific Device not found

Deletion of entities

The following syntax is used for any delete operation on the device restriction model:

<deleteFromDeviceRestrictionModel> ::= "DELETE" "FROM" "RESTRICTION" |
"DEVICE_CLASS" | "SPECIFIC_DEVICE" "WHERE" <term>

Summary

We presented syntax to modify and create organization hierarchies and device restrictions. Based on a organization model thus created, the staff assignment query can be used to assign staff members to activities.

5.3.2 Data Type and Constraint Modeling

Data types are declared with the following syntax. Three operations are possible: create, modify and delete. Create means the creation of a new data type, modify the modification and delete, its deletion.

The data type definition from sec. 5.2.3 is used when defining new data types or altering existing ones.

Create syntax

The following syntax defines, how new data types are creates:

<createDataType> ::= "CREATE" "DATATYPE" *name* <type>

Possible failures:

TYPE_ALREADY_EXISTS	The data type cannot be created because a data type with *name* already exists.

Rename

The following syntax defines, how data types are renamed:

<alterDataType> ::= "ALTER" "DATATYPE" *oldname* "RENAME" *newname*

Possible failures:

TYPE_ALREADY_EXISTS	The data type cannot be renamed because a data type with the *newname* name already exists.

Modifying the data type and setting a new data type definition

The following syntax defines, how a data type's type definition is modified.

<modifyDataType> ::= "ALTER" "DATATYPE" *name* <modifyType>
<modifyType> ::= "MODIFY" "TYPE" <type>

No operation specific error codes.

Redefining subtypes

Struct data types possibly have a hierarchical structure. To change a struct element's data type, the following syntax is defined:

<alterDataSubType> ::= "ALTER" "DATATYPE" *name* <alterSubType>
<alterSubType> ::= "MODIFY" "SUBTYPE" <subtype> "(" *data type definition* ")"
<subtype> ::= *name* ["."<subtype>]

The `subtype` is used to navigate to the elements embedded in structs. It is also allowed to thereby navigate to another type.

Possible failure reasons:

SUBELEMENT_NOT_FOUND	The subtype does not exist

Example: Struct S has an element a of type Struct Y, which has an element b. It is possible to modify the type for element b to Text by accessing Struct S:

ALTER DATATYPE S MODIFY SUBTYPE a.b (TEXT)

5.3.3 Process Template and Process Instance Modeling

Sec. 5.2.7 introduced a Formal Process Fragment Representation that is heavily used in this chapter. The Process Fragment Representation is fully capable of representing a process model by means of storing a Process Fragment under a name.

CREATE Syntax

Process templates are created via a create syntax:

<createProcessTemplate> ::= "CREATE" "PROCESS_TEMPLATE" *name* ["(" <pfr> ")] ["WITH "START_PARAMETERS" "(" <Struct> ")"] ["WITH" "RETURN_PARAMETERS" "(" <Struct> ")"]

The `<read>` declarations of the `<pfr>` express the input parameters and their mapping to data containers, analogue for return parameters.

Retrieving process fragments

The `<printPfr>` syntax selects a process fragment from a process model:

<printPfr> ::= "PRINT_PFR" "OF" ["TEMPLATE"|"INSTANCE"] ["FRAGMENT" "=" *fragmentname*]

If the `fragment = fragmentname` syntax is not used, the root fragment is selected.

Possible failure reasons:

FRAGMENT_NOT_FOUND	No fragment was found for the given name

Positive answers are expressed through the Process Fragment Representation.

Example:

PRINT_PFR OF TEMPLATE "OrderProcess" fragment = "Order"

The example returns the fragment named "Order" with all its sub structures in the Process Fragment Representation syntax.

Pattern AP1: INSERT Process Fragment

The serial insert inserts the specified Process Fragment after or before the named ↪*fragment* .

<ap1InsertProcessFragmentSerialInsert> ::= "INSERT" "ROOT"|"AFTER"|"BEFORE" ("PROCESS_TEMPLATE" *name*) | ("PROCESS_INSTANCE" *name*) | ("PROCESS_INSTANCES" (*name* +)) ["FRAGMENT" "=" ↪*fragment*] <pfr>

If no root process fragment exists, "INSERT ROOT" shall be used without specifying the fragment. If after or before is used, the fragment has to be specified. Before and after require the parent fragment of the fragment that was named to have the ability, to store fragments before it.

Possible failure reasons:

ROOT_FRAGMENT_EXISTS	A root fragment already exists
FRAGMENT_NOT_FOUND	The named fragment(s) could not be found in the process template /instance
FRAGMENT_MALFORMED	the given fragment is not sound
DATA_ELEMENT_NOT_FOUND	referred data elements do not exist in the process template / instance
TYPE_MISMATCH	data types mismatched
SYNCHRONIZATION_EDGE_MISSING	a synchronization edge is missing

Inserting alternative control flows

Other than specified in the referred Change Patterns [WRS08], no “parallel Insert” and “conditional Insert” is declared for the WfQL, but an insert of split and loop nodes around process fragments. This approach is just as powerful and greatly reduces the required number of different operations.

The following definition is applicable for XOR, OR and AND blocks. The `BRANCH` definition is used for `AND` blocks. `OR` and `XOR` blocks use `CASE` and have to state a `decisionPredicate`.

<ap1InsertProcessFragmentAlternativeFlow> ::= "INSERT" "INTO" ("PROCESS_TEMPLATE" *name*) | ("PROCESS_INSTANCE" *name*) | ("PROCESS_INSTANCES" "(" *name* + ")") "FRAGMENT" "=" ↪*fragment* ("CASE" <decisionPredicat> ":") | ("BRANCH" ":") <pfr>

Possible failure reasons:

FRAGMENT_NOT_FOUND	The named fragment(s) could not be found in the process template /instance
FRAGMENT_MALFORMED	the given fragment is not sound
DATA_ELEMENT_NOT_FOUND	specified data elements do not exist in the process template / instance
TYPE_MISMATCH	data types mismatched
SYNCHRONIZATION_EDGE_MISSING	a synchronization edge is missing

Pattern AP2: DELETE Process Fragment

<ap2DeleteProcessFragment> ::= "DELETE" "FROM" ("PROCESS_TEMPLATE" *name*) | ("PROCESS_INSTANCE" *name*) | ("PROCESS_INSTANCES" "(" *name* + ")") ["FRAGMENT" "=" ↪*fragment*]

Deletes the named process fragment from the process template or instance. If it was surrounded by a case in a control flow, the case is deleted, if it contains no control flow any more. If no fragment is named, the root is chosen.

Possible failure reasons:

FRAGMENT_NOT_FOUND	The named fragment(s) could not be found in the process template /instance
BLOCK_CONSTRAINTS	node1 and node2 are not nested on the same block level

Pattern AP3 / AP4: MOVE or SWAP Process Fragment

The following definition expresses the move, swap and move into operations on process fragments of the same process model.

When using the move into operation, the case or branch of the surrounding block has to be named depending on the type of the surrounding block, analogue to Pattern AP1.

<ap3MoveProcessFragmentALTER> ::= "ALTER" ("PROCESS_TEMPLATE" *name*) | ("PROCESS_INSTANCE" *name*) | ("PROCESS_INSTANCES" "(" *name* + ")") "MOVE" ↪*selectedFragment* " "AFTER"|"BEFORE"|"SWAP"|"INTO" ↪*targetFragment* [("CASE" <decisionPredicat>) | "BRANCH" "]

Possible failures:

TARGET_FRAGMENT_NOT_FOUND	The named target fragment could not be found in the process template /instance
FRAGMENT_MALFORMED	the given fragment is not sound
DATA_ELEMENT_NOT_FOUND	specified data elements do not exist in the process template / instance
TYPE_MISMATCH	data types mismatched
SYNCHRONIZATION_EDGE_MISSING	a synchronization edge is missing
WRONGFUL_INSERT	Inserting with CASE or BRANCH does not reflect the type of the block it was inserted into

Pattern AP4: REPLACE Process Fragment

The replace syntax replaces the selected process fragment with the declared process fragment:

<ap4ReplaceProcessFragment> ::= "ALTER" ("PROCESS_TEMPLATE" *name*) | ("PROCESS_INSTANCE" *name*) | ("PROCESS_INSTANCES" (*name* +)) "REPLACE" ↪*fragment* "WITH" <pfr>

Replaces the *fragment* with the `<pfr>`

Possible failures:

FRAGMENT_MALFORMED	the given fragment is not sound
DATA_ELEMENT_NOT_FOUND	specified data elements do not exist in the process template / instance
TYPE_MISMATCH	data types mismatched
SYNCHRONIZATION_EDGE_MISSING	a synchronization edge is missing

Altering the process definition

The process definition of Process Instances and Process Templates can be altered with the following operations:

Altering Process Fragments

A Process Fragment may have a number of standard properties, such as:

- data container definitions
- read and write mappings
- exception blocks
- process fragment (depending on the type of process fragment: multiple ones in branches or cases)

The WfQL offers a number of alteration operations to modify the attributes of individual process fragments:

Alter Data Container definitions

The following syntax alters data container definitions: Renames the data container, redefines data containers, adds and removes them:

<alterDataContainer> ::= "ALTER" "DATA" "OF" ("PROCESS_TEMPLATE" *name*) | ("PROCESS_INSTANCE" *name*) | ("PROCESS_INSTANCES" "(" *name* + ")") "FRAGMENT" "=" *fragment* ("RENAME" *oldName* TO *newname*) | ("REDEFINE" *containerName* "WITH" <type>) | ("REMOVE" *containerName*) | ("ADD" *containername* "WITH" <type>)

Alter read and write mappings

The following syntax alters read and write mappings of process fragments: Changes their source or target, removes them and inserts new mappings:

<alterReadWrite> ::= "ALTER" "READ"|"WRITE" "OF" ("PROCESS_TEMPLATE" *name*) | ("PROCESS_INSTANCE" *name*) | ("PROCESS_INSTANCES" "(" *name* + ")") "FRAGMENT" "=" *fragment* ("CHANGE" "SOURCE"|"TARGET" *oldReference* "TO" *newReference*)|("REMOVE" *sourcename* "TO" *targetname*)|("INSERT" *sourcename* "TO" *targetname*)

Alter exception blocks

The following syntax alters exception blocks of process fragments: Replaces, adds and removes them.

<alterExceptionBlocks> ::= "ALTER" "EXCEPTION" "OF" ("PROCESS_TEMPLATE" *name*) | ("PROCESS_INSTANCE" *name*) | ("PROCESS_INSTANCES" "(" *name* + ")") "FRAGMENT" "=" *fragment* "," "type" "=" *exceptionName* ("REPLACE" "WITH" <pfr>| "REMOVE"|"ADD" <pfr>)

5.3.4 Process Instance Modeling

In sec. 5.3.3 we already defined, how to change the Process Model of Process Instances. At times, Process Instances have to be created, that are not based on a Process Template. The following syntax definition describes, how to accomplish this:

The following syntax creates a new process instance with empty root pfr and gives it a (unique) name:

CREATE PROCESS_INSTANCE *name*

Possible failures:

UNIQUE_NAME_CONSTRAINT	The name was already used for a process instance

The resulting process instance is accessible with the name supplied. This differs from the instantiation of process templates, where process instances are not named.

Further change modeling is analogue to sec. 5.3.3.

5.3.5 Activity Template Modeling

Activity templates reside in the `ACTIVITY_TEMPLATES` entity set and are created and altered by the following syntax.

CREATE Syntax

The following create syntax is used to create new activity templates:

```
<activityStartParameterSignature> ::= <Struct>
<activityStopParameterSignature> ::= <Struct>
<activityTemplateCreate> ::= "CREATE" "ACTIVITY_TEMPLATE"  name  "("
<activityStartParameterSignature>"," <activityStopParameterSignature>"," "("
exceptionName + ")" ")"
```

`activityStartParameterSignature` declares a Struct Data Type, which elements are the parameter names, analogue for `activityStopParameterSignature`. `exceptionName` references allowed Exception Types by name.

Possible failures (in addition to data type declaration failures):

EXCEPTION_UNKOWN	one or more exceptions in *`exceptionName`* is unknown
ACTIVITY_TEMPLATE_EXISTS	an activity template with this name already exists

ALTER Syntax

The alter syntax allows for alterations of activity templates.

Alter Start Parameter Signature

The alter start parameter syntax changes the start parameter of a named activity template by use of the `modifySubType` syntax:

```
<activityTemplateAlterStartParameters> ::= "ALTER" "ACTIVITY_TEMPLATE"  name
"STARTPARAMETER" <modifySubType>
```

Analogue to `activityTemplateAlterStartParameters`:

<activityTemplateAlterReturnParameters> ::= "ALTER" "ACTIVITY_TEMPLATE" *name* "RETURNPARAMETER" <modifySubType>

Possible failures (in addition to modifySubType and data type failures):

ACTIVITY_TEMPLATE_NOT_FOUND	no activity template found under the given name

Add exception to Activity Template

The following syntax adds *exception_name* to the set of possible exceptions for the activity template named:

<activityTemplateAlterAddException> ::= "ALTER" "ACTIVITY_TEMPLATE" *name* "ADD_EXCEPTION" *exception_name*

Possible failures:

ACTIVITY_TEMPLATE_NOT_FOUND	no activity template found under the given name
EXCEPTION_ALREADY_EXISTS	Exception already part of the set of allowed exceptions

Remove exception from Activity Template

The following syntax removes *exception_name* from the set of possible exceptions for the activity template named:

<activityTemplateAlterAddException> ::= "ALTER" "ACTIVITY_TEMPLATE" *name* "REMOVE_EXCEPTION" *exception_name*

Possible failures:

ACTIVITY_TEMPLATE_NOT_FOUND	no activity template found under the given name
EXCEPTION_DOES_NOT_EXIST	Exception is not part of the exception list or does not exist

DELETE Syntax

The delete syntax for activity templates deletes an activity template:

<activityTemplateDelete> ::= "DELETE" "ACTIVITY_TEMPLATE" *name*

Possible failures:

ACTIVITY_TEMPLATE_NOT_FOUND	no activity template found under the given name
ACTIVITY_TEMPLATE_IN_USE	activity template is in use in a process instance or process template or by an activity instance.

5.4 Execution Modules

The Execution Modules cover the Execution Stage of a Process's Life-cycle. They cover the syntax for gaining access to Activity Instances, instantiating Process Instances, starting, pausing, stopping and aborting Process Instances and the registration of Monitors and the transport of notifications to clients.

5.4.1 Activity Execution

Activity executing stakeholders start activities by checking them out. This is comparable to a version control system, that allows just one stakeholder to check out a specific file or part of a file. Analogue, an Activity Instance can get checked out just by one stakeholder.

To do so, the stakeholder sends the following command:

<activityCheckout> ::= "CHECKOUT" ↪*activity_instance*

Success response:

<checkoutResponse> ::= "AOK" "CHECKOUT" "(" "startParameters" ":" *start_parameters* "," "startParameterSignature" ":" *start_parameter_type_definition* "," "returnParameterSignature" ":" *return_parameter_type_definition* "," "exception_set" ":" *exception_names* ")"

Possible failures:

ACTIVITY_INSTANCE_NOT_AVAILABLE	Activity instance is not available for checkout – possibly checked out by another stakeholder
ACTIVITY_INSTANCE_UNKNOWN	Activity instance is unknown

Yield

When an activity instance cannot be executed due to a problem, before the real world process was altered, it can be yielded. After a yield, it can be checked out by another stakeholder.

<yieldActivityInstance> ::= "YIELD" "ACTIVITY_INSTANCE" ↪*activity_instance*

Possible failures:

ACTIVITY_INSTANCE_UNKNOWN	Activity instance is unknown

Checkin

Checkin is the act of returning an activity instance with return parameters after a successful execution:

<activityCheckin> ::= "CHECKIN" "ACTIVITY_INSTANCE" ↪*activity_instance* "(" *return_parameters* ")"

Possible failures:

ACTIVITY_INSTANCE_UNKNOWN	Activity instance is unknown

Exception handling

When the stakeholder encounters a failure that it cannot solve, it throws one of the allowed exceptions that it received with the checkout.

<throwException> ::= "THROW" "EXCEPTION" "FOR" ↪*activity_instance* "(" *exception_type* "," *serialized exception* ")"

The *serialized exception* contains the serialized exception that reflects the data type definition of the *exception_time*.

Possible failures:

EXCEPTIONTYPE_UNKNOWN	Exception type is unknown
ACTIVITY_INSTANCE_UNKNOWN	Activity instance is unknown

Accessing the Worklist

Activity Instances get assigned to one or many stakeholders. For stakeholders to checkout activity instances and process them, they need to gain access to the identifier (reference) of activity instances. Thus, each stakeholder can access a so called *worklist* which is a per stakeholder view on currently ready-to-run activity instances.

<accessWorklist> ::= "SHOW" "WORKLIST" ["WHERE" <term>]

The WHERE <term> limits the result-set. The <term> operates on the attributes of all accessible activity instances.

The WfMS responds with tuples of the following structure:

activity instance ID	activity template name

5.4.2 Process Execution

This section covers the execution and control of process instances: The instantiation of process instances, the control over the execution (start, stop, pause and abort) and the modification of running process instances.

Instantiate Processes

Creates a Process Instance based on a Process Template – if start parameters are stated, they have to be supplied.

<instantiateProcessTemplate> ::= "INSTANTIATE" "PROCESS_TEMPLATE" ↪*process_template* ["WITH" "PARAMS" (*start_parameters*]

Possible failures:

PROCESS_TEMPLATE_UNKNOWN	Process template unknown
DATA_TYPE_CONSTRAINT	Start parameters are not of the correct type or violate the data type constraints

If executed successfully, the WfMS returns the process instance's identifier to the client:

<instantiateProcessTemplateResponse> ::= "AOK" "PROCESS_INSTANCE" "WITH" "ID" ↪*process_instance*

After instantiation, the process instance is not started automatically but has to be started via the `startProcessInstance` operation. Therefore, it is possible to instantiate a process template, modify the instance and start it.

Start Process Instances

Starts non-running process instances.

<startProcessInstance> ::= "START" "PROCESS_INSTANCE" "WITH" ("name" "=" *name*) | ("ID" "=" ↪*Process_Instance*)

Possible failures:

PROCESS_INSTANCE_UNKNOWN	Process instance unknown

Control Execution of Process Instances

Process instances can be paused, stopped or aborted.

Pause means, the Execution engine of the Workflow Management System will not continue the Process Model's execution. *Stop* means, that the WfMS will try to notify all activity executing stakeholders that they should pause their execution. *Abort* means, that the process instance is aborted – all stakeholders get notified if possible. If not, their return of activity instances will have no effect.

<controlProcessInstance> ::= "PAUSE" | "STOP" | "RESTART" | "RESUME" | "ABORT" "PROCESS_INSTANCE" ↪*process_instance*

Possible failures:

PROCESS_INSTANCE_UNKNOWN	Process instance unknown

Updateing Process Instance's Data Containers

Replace a Process Instance's Data Container value with a new one.

<modifyDataContainers> ::= "UPDATE" "PROCESS_INSTANCE" ↪*process_instance* "DATACONTAINER" "SET" *dataContainerName* "=" *new_value* ["FRAGMENT" = *fragmentName*

If the fragment is named, the local data container of the named fragment is modified.

Possible failures:

PROCESS_INSTANCE_UNKNOWN	Process instance unknown
FRAGMENT_UNKNOWN	Process fragment not found in process model
DATA_ELEMENT_UNKNOWN	Data Element unknown
VALUE_TYPE	Value is not of the type of the data element

Update start parameters of Activity Instances

Update the start parameters of an existing activity instance with new start parameters. Has to be applied before the activity instance was checked out.

<modifyActivityInstanceStartParameters> ::= "UPDATE" "ACTIVITY_INSTANCE" ↪*activity_instance* "SET" "STARTPARAMETER" "=" *new_value*

Possible failures:

ACTIVITY_INSTANCE_UNKNOWN	Activity instance is unknown
DATA_TYPE_CONSTRAINT	Value is not of the type of the start parameters.
WRONG_ACTIVITY_INSTANCE_STATUS	Activity instance was already checked out (currently getting processed or returned)

5.4.3 Event registration and delivery

Events inform stakeholders about changes in the Workflow Management System. It is intended as delivery method for notifications about ready-to-run activity instances and process termination.

Stakeholders register **monitors** for the events they desire to get notified about. The registration is bound to the current session, which means, that even if the client-server connection drops, but the session is still active, the events would get cached until the session is reestablished.

An event is always triggered and delivered. Even when the event's information is no longer current, the stakeholder still gets informed. For example, this could mean, that a stakeholder gets informed about a new activity instance's ready-to-run state. But the stakeholder does not get informed when another stakeholder checks out the activity instance.

Activity Instance Events

A stakeholder may register a monitor to a class of activities to get informed about newly ready-to-run activity instances. A class may cover all activities possibly executed by the stakeholder or cover a filtered subset of the activity instances, filtered per activity templates, process template or process instance.

Thus, we define the registration command:

<registerActivityInstanceMonitor> ::= REGISTER MONITOR ON ACTIVITY_INSTANCES [WHERE <term>]

In order to formulate a filter, the general `WHERE`-clause is used.

No special failure codes exist.

Events are delivered with the following syntax:

<activityEvent> ::= EVENT ACTIVITY_INSTANCE "{" "date" ":" *date* "," "aiD" ":" *activity instance ID* "," "atID" ":" ↪*activity template ID* }

Process Instance Events

At times, stakeholders are interested in getting notified about certain process instance's state. The following commands allow for the registration and reception of termination events, that are created when a Process Instance gets terminated.

```
<registerProcessInstanceMonitor> ::= REGISTER MONITOR ON
PROCESS_INSTANCES [ WHERE <term> ]
```

Analogue to the Activity Instance monitor, the `WHERE <term>` is used.

Events are delivered with the following syntax:

```
<processEvent> ::= EVENT PROCESS_INSTANCE "{" "date" ":" date ,
"process_instance" ":" ↪process_instance "}"
```

Listing and deleting monitors

A stakeholder may require access to the list of registered monitor.

Listing monitors Monitors are listed with the following syntax:

```
<listMonitors> ::= SHOW MONITORS
```

The result-set contains entities with the following entries:

monitor-type	monitor-filter
monitor-type	either "processInstance" or "worklist"
monitor-filter	the term, that was used in the `WHERE` clause. Can be null.

Deleting monitors A monitor is deleted by stating its monitor-type and monitor-filter:

```
<deleteMonitors> ::= "DELETE" "MONITOR" "WITH" "TYPE" "="
"processInstance"|"worklist" "," "FILTER" "=" <term>
```

Possible failures:

NO_MONITOR_MATCHED	No monitor was matched with these properties

5.5 Analysis Modules

The analysis modules category contains modules, that allow analysis of process instances and activities. It is split into the analysis of process instances and the analysis of activities and their process model.

5.5.1 Accessing Process Instances

The values stored in data containers of process instances can be accessed, as well as the status of fragments inspected by use of the following syntax.

Inspecting data elements of Process Instances

The following syntax reads the data container of process instances. If the fragment is named, the local data context of it will be searched. The `WHERE` clause specifies, which process instance(s) are inspected:

<inspectDataElement> ::= "SELECT" *dataContainer* "FROM" "PROCESS_INSTANCE" ["FRAGMENT" = *fragmentName* ["WHERE" <term>]

Possible failures:

DATA_CONTAINER_NOT_EXISTING	Named Data Container not found
FRAGMENT_NOT_FOUND	The process fragment could not be found.
INSTANCE_NOT_FOUND	The process instance could not be found.

Inspect a Process Instance's Progress

The progress of a process instance is expressed by the status of its process fragments. The following syntax is used to inspect the status:

<inspectNodeStatus> ::= "SELECT" "STATUS" "FROM" "PROCESS_INSTANCE" "FRAGMENT" = *fragmentName* ["WHERE" <term>]

Returns the status of the named fragment.

Possible failures:

FRAGMENT_NOT_FOUND	The process fragment could not be found.
INSTANCE_NOT_FOUND	The process instance could not be found.

5.5.2 Activity and Process Analysis

Activity and process analysis covers the analysis of activity instances and their related process model.

Inspecting Activity Instances

This section covers the analysis of activity instances: Their start and return parameters and their status.

Inspecting Start Parameters and Return Parameters of Activity Instances

The following code shows, how to inspect the start and return parameters of activity instances. If the return parameters were not yet set or the activity instance was terminated with a failure, they return a null value.

<modifyActivityInstanceStartParamters> ::= "SELECT" "START_PARAMETERS" | "RETURN_PARAMETERS" "FROM" "ACTIVITY_INSTANCES" ["WHERE" <term>]

Possible failures:

ACTIVITY_INSTANCE_UNKNOWN	
Activity instance is unknown	

Server returns a set of tupels with the start parameters and return parameters respectively.

Inspecting Activity Instance State An activity instance can be in one of five states: Ready, Checked-out, Checked-in, Finished, Paused and Aborted. In order to read the state

of activity instances, the client executes:

<selectActivityInstanceState> ::= "SELECT" "STATUS" "FROM"
"ACTIVITY_INSTANCES" ["WHERE" <term>]

The server returns a `LIST` of one-value tupels that contain the state.

Accessing the process model via Activity Instances

The process fragment, that triggered the creation of the activity instance can be accessed via the following syntax:

<selectActivityInstanceModel> ::="SELECT" "activity_node" "," "process_instance"
"FROM" "ACTIVITY_INSTANCES" ["WHERE" <term>]

The server returns a `LIST` of tuples with the activity process fragment and the process instance of the activity instances.

Using the Select module for analysis purposes

The syntax of this module is heavily making use of the “Select module”. The following definitions are directly derived from the realized WfMS model from chapter 4:

- `selectActivityInstanceState`
- `selectActivityInstanceModel`
- `modifyActivityInstanceStartParamters`
- `inspectNodeStatus`

All attributes shown in the WfMS model can be accessed, therefore further analysis capabilities that are not explicitly stated in the analysis modules exist. Furthermore, due to possibility to recursively reuse the result-sets of WfQL operations, data intensive queries that would be used as input for further queries with smaller result-sets can executed directly.

For example, to print the Process Fragment, that triggered the instantiation of an activity instances, the following query can be used:

```
PRINT_PFR OF INSTANCE fragment = (SELECT activity_node FROM
ACTIVITY_INSTANCES WHERE ID = 1) WHERE ID = (SELECT process_instance
FROM ACTIVITY_INSTANCES WHERE ID = 1 )
```

5.6 Summary

In summary, we presented the language definition for the WfQL, that is capable of representing process models, the modeling of process models, the execution of processes and the analysis of running and past ran processes.

6 Implementation

In this chapter, several prototypes are presented, that implement interfaces using the WfQL: a WfMS server with a WfQL server interface and a number of client programs. Because of several new aspects of the WfQL, that cannot be found in current WfMS implementations, we could not make use of existing WfMS as back end, but instead, implemented a WfMS from scratch.

We implemented

Three classes of clients were written:

1. A Python Webserver, that instantiates Processes with start parameters based on HTTP requests, waits for the termination and delivers the return parameters to the Browser.
2. A Python Client, that waits for activities of a certain activity template and executes them
3. A Java SWT Client used for modeling process templates.

All clients target different aspects of the language. While the webserver is primarily concerned with starting processes and waiting for their termination, the python client automatically (without user interaction) executes activities. The Java SWT Client can be used to model process templates.

6.1 Server

The prototypical server implementation offers a subset of the WfQL's defined syntax and semantic.

We have restricted the syntax to a limited subset of the modeling capabilities: insert and delete operations. A subset of the possible Process Fragment: Activities, Serial Blocks and

AND/AND Blocks. Text and Integer basic data types are supported, as are Structs. The data flow is fully implemented, as is the activity execution.

The actual implementation was written in Java 1.6, targeting Java Application Servers. We used the following Libraries:

- Spring 3.0 [Spring]
- Hibernate 3.4 [Hib]
- Stringtree-JSON 2.0

The latter library is used to serialize and de-serialize data per defined in data serialization in sec. 5.2.4. Spring and Hibernate are used, as they offer distributed, fast and scalable support for transactions and data storage. For the actual deploy, we used the JBoss Application Server 5.0 GA, however, it should be deployable on any JavaEE compatible application server as well.

6.1.1 Architecture

The server architecture can be split into four components as shown in Figure 6.1: A TCP server, that accepts TCP connections and creates a new thread for each connection. A parser, that is used by the TCP Server to parse incoming operations and to generate WfQL-conform output. The parser in turn uses the WfMS Engine component to apply changes to the WfMS based on the parsed operations. The WfMS Engine uses the Hibernate Library to persist its objects into a relational database management system. The wiring between the components is configured via the spring library, that is executed on deploy on the application server and handles the actual deploy and start of our components.

Data storage

Figure 6.2 shows the entity relationship between the workflow management entities used in the prototypical implementation.

Entities like: activity template and process model define data types stored in data type definition objects. Data type definition objects are capable of expressing basic data types like Integer and Text and Structs, which contain key value pairs, the key being a Text, the

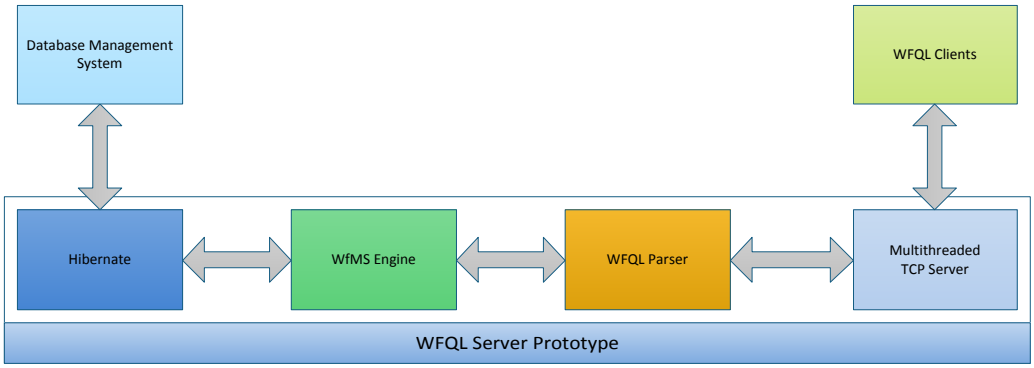


Figure 6.1: Server architecture

value a data type definition entity and thus define structs as defined in the realization and language definition.

Any data represented in Figure 6.2 and data type definitions are parsed based on the language definition in the parser, and generate corresponding data storage objects, that can be stored via hibernate into a relational database management system. The parser internally works with the data storage objects and parses the WfQL with a as late as possible scheme to data storage objects: That means, that if a data storage object such as a Process Fragment Representation contains attributes such as `READ` or `WRITE` mappings, the Process Fragment Representation (Pfr) is created (but not stored), the mapping is parsed, created and later added into the Pfr. Ultimately, the data is persisted. If a parsing error occurs in the meantime, the data is discarded and the appropriate error is returned via the parser.

6.1.2 Data Types

The Data Types are implemented by means of persistable objects that are generated by means of parsing the WfQL's Data Type Definition. The prototypical implementation implements `Text` and `Integer` basic data types and the `Struct` data type. The generated objects reflect the flexibility of the Data Type Definition and can be extended, based on the existing code, to accomodate other types like `List`, `Set` and `Map`.

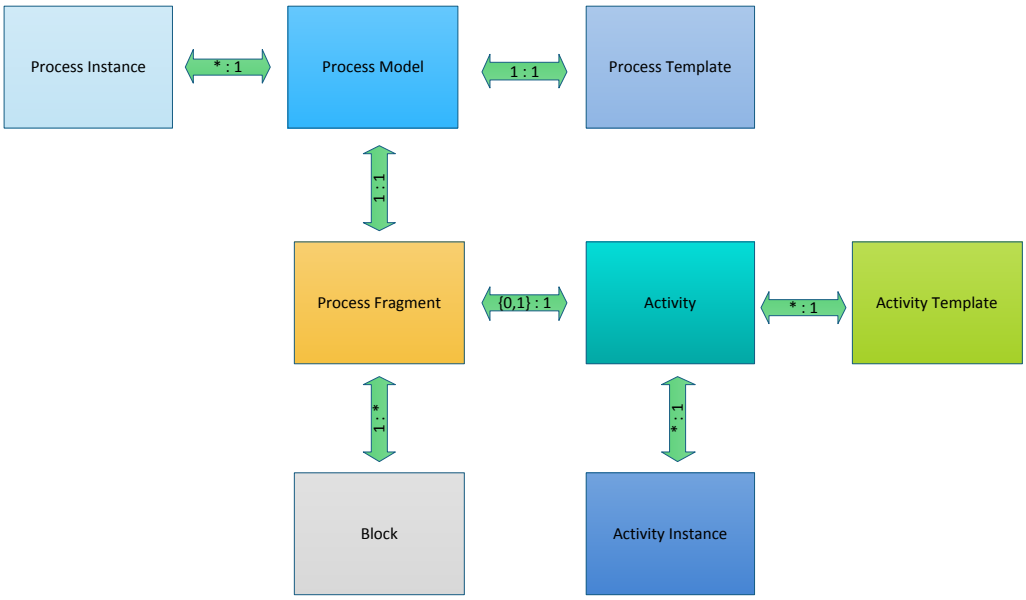


Figure 6.2: Persistence data model

6.1.3 Parsing the model

Analogue to the Data Type Definition, parsing the Process Fragment Representation is implemented with a recursive parser that returns Process Fragment Representation objects. These PFR objects are persistable and are used in the Process Model.

6.1.4 Data flow

The data flow is expressed by data containers and the `READ` and `WRITE` mapping. Each block has a local data container context. While parsing the model, data containers from outer blocks are carried into the local data container of inner block(s). This is implemented by recursively iterating through blocks, carrying a map of data containers. If a block redefines an already taken name, it overwrites this data container in the local data container.

When the execution engine reads data via the `READ` mapping, it accesses the local data context and thus per design accesses the correct data container. Analogue for writing, because data containers are never overwritten, but the value is written inside of them. That means, even when activities return data in the form of return parameters, that gets serial-

ized into data containers, these data containers are not replacing the local data context of blocks when written, but their values are copied over instead.

6.1.5 The WfMS Engine

We implemented a minimalistic WfMS Engine to test the parser, and to offer a functional prototype with WfMS properties. Although a great number of WfMS Engines exist, we had unique requirements that are not satisfied by existing solutions. Therefore, we decided to write a WfMS from scratch, that the parser uses:

The engine is event driven. At no point, it acts on a timer or repeatedly queries the database for new information. Even though we did not explicitly state event drivenness as a requirement, it is much easier to satisfy the concurrency requirements. The alternative, a non-event driven engine that periodically checks for pending tasks could not abort a transaction if it would violate the constraints given by the meta model and would thus yield an error that cannot directly be traced to the originator. Even worse, the originator cannot directly be informed about the failure, when executing an operation that causes a problem.

The entire concurrency model as defined in sec. 3.2 cannot easily be satisfied when implementing a non-event driven system.

As a result, when a stakeholder starts a process instance, the system instantiates the activities given by the model, that have no dependencies on other activities. The system returns only after the instantiation was successful. If the start parameters would not satisfy the activities, the activity instantiation would fail, which would cause the start of the process instance to fail as well.

6.1.6 Deployment of the prototype

The source code of the prototypical server implementation and deploy instructions can be found on the supplied data medium in the following directory:
`/implementation/server/`

To execute it, a Java Runtime Environment of at least Version 1.6 is expected, as well as a JavaEE 1.6 compatible application server. For persistance, a postgres database version

8.3 is used. However, the deploy instructions describe, how to generate initial databases for other SQL dialects and how to configure the server accordingly.

6.1.7 Summary

We developed a prototypical implementation of a WfMS server with a WfQL interface. The focal points for us are the parser and the WfQL. The source code is documented and we hope, that in case the WfQL is extended or further works based on it, that the prototype is a good starting point for new implementation.

Java source code lines: 4266
Size in bytes of Java source code: 136843
XML Configuration code lines: 853
Size in bytes of XML Configuration: 32913

6.2 Modeling tool

The prototypical modeling tool's intent is to show, how modeling tools interface with a WfQL capable WfMS. It supports the following operations: The creation of new process templates, viewing process templates and inserting and deleting of process fragments of type activities, serial blocks and and/and blocks.

Figure 6.3 shows a screenshot of the modeling tool's primary window with a Process Template.

6.2.1 Data storage

The modeling tool stores one template at a time. It shows the user a list of available templates. The user selects one, which triggers a download of the selected process template's model which is temporarily stored in memory. The persistence is realized through the WfQL interface of the WfMS.

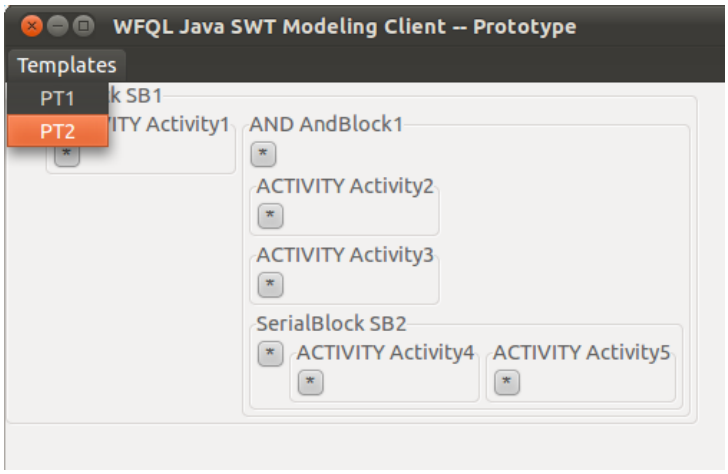


Figure 6.3: Modeling tool screenshot

Internally, the object oriented Java programming language allowed us to use inheritance to store the model as shown in Figure 6.4. The `Pfr` (Process Fragment Representation) class is inherited by an activity class, that implements the activity specific properties. The `Block` class implements block specific properties, which are fine granulated in the `SerialBlock` and `AndBlock` classes.

The modeling tool implements a parser for the WfQL's Process Fragment Representation (sec. 5.2.6). Models are persisted when the user changes them via the gui. To do so, any part (including the root) of any pfr based class can be exported into the Process Fragment Representation syntax. The modeling tool uses the WfQL to apply changes to the model and to load existing templates.

6.2.2 Usage

The supplied data medium contains binaries, source code and start instructions that can be found in the following directory:
`/implementation/client/modeling_tool/`

Java source code lines: 1062
Size in bytes of Java source code: 26961

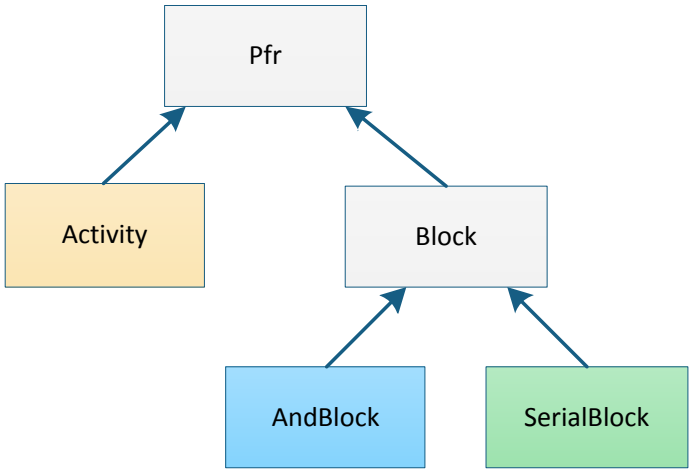


Figure 6.4: Modeling tool inheritance model

6.3 Web-services a lá Workflow

The hierarchical and strict block structure of the underlying Adept meta model allows for the design of more than just workflows. With the implementation of a decision logic and hierarchical data containers, a programming meta model was implicitly designed. The Web-service implementation that we are presenting makes use of this programming meta model to deliver Websites. The content is created by any number of activity programs and delivered by a Web-server The latter instantiates processes from process templates with start parameters based on parameters received from browsers.

In addition to the capability of generating content with any number of languages and on different platforms, we implicitly created a cluster (or cloud) of execution instances, that can act as a fail-over cluster or cloud.

Depending on the implementation of the WfMS, this swarm of activity programs can be expanded and used to easily make web services or any information processing, that is process oriented, scalable.

6.3.1 Components

We identify three distinct components of a Web-service with WfMS based content generation:

1. The WfMS Server
2. The Web-server
3. Content generating and processing activity programs

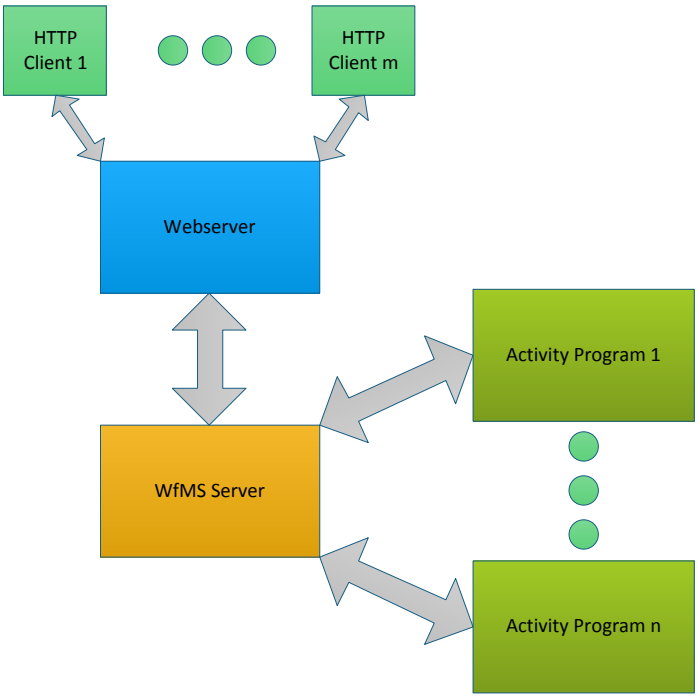


Figure 6.5: Webservice a lá workflow overview

In our example implementation, the Webserver makes use of predefined Process Templates. When a request for a web-page arrives at the web-server, it creates a new process instance with start parameters, based on the request's parameters. The web-server waits for the process instance to terminate and delivers the return parameters as web-page.

The process is modeled, so activities create the content for the web-page and return them in a specified return parameter.

The content generating and processing activity clients process their parameters and create content based on them, which they return to the process instance.

Therefore, in order to deliver a web-page, the following steps are necessary:

- Instantiation of a Process Template based on the URI
- Execution of the Process Instance with Content generation by activity clients

- Acquisition of return parameters of the corresponding process instance by the Webserver and delivery of the return parameters as a web page to the browser.

6.3.2 Webserver component

The webserver component is written in the Python programming language with a minimalistic, single threaded http webservice. It opens a TCP connection to the WfMS and uses the WfQL to communicate the instantiation of process templates and waits for the termination of the instances to deliver the return parameters to its client.

Total lines of python code: 84
Total size of python code in bytes: 2575

6.3.3 Activity client component

The activity client component is written in the Python programming language as well, and is actually more complex than the server component. It periodically checks for newly available activity instances, makes sure it is capable of executing the activity template of the available activity instance, checks the instance out, executes the task in question and checks the activity back in.

Total lines of python code: 94
Total size of python code in bytes: 2653

6.3.4 Usage

Source code, start instructions can be found on the supplied data medium in the following directory:
`/implementation/client/webservice/`.

6.4 Summary

This chapter showed a number of applications that use the WfQL each in their own context. The prototypical server implementation offers access to its WfMS capabilities via the WfQL. The modeling tool uses this WfMS via the WfQL to load and store process models. The webservice component starts processes, waits for their termination and gathers return parameters. The activity client component automatically executes activities whose activity template it knows. We thus covered three of the four stages of the process life-cycle: Modeling, Execution and Change. The analysis stage was tested manually by connecting to the WfMS via telnet.

We thus come to the conclusion, that the WfQL can be used to interconnect components of a WfMS.

7 Discussions and possible future works

This chapter discusses the extension of the WfQL and WfMS with further concepts that have not been further discussed in the main chapters of this work. We think it is necessary to discuss them here, although the concepts are incomplete and require further research.

7.1 Testing paths in process models

The combination of Data Types and Data Constrains – available both on process model level and data type declaration level – offer limiting factors to any data hold in the data containers of a process instance. These constrains are known when modeling the process model, which would allow for testing of the decision predicates of process models and predetermine which paths could be chosen during the runtime of a process instance that is based on the model.

This allows in turn for simpler decision predicates. Given, that a data container is of type Integer and has the constraint `Range` between 1 and 3 set on data type definition level on it, a decision predicate evaluating the data container only has to evaluate the values 1,2, and 3. Therefore, a decision predicate evaluating the data container *InputValue* with three separate tests, one for each possible value, will always evaluate to true for one of the three possibilities.

Therefore, a extensive model check is possible via the combination of data types and data constraints.

7.2 Data transport protocol – discussing the serialization method

In sec. 5.2.4 we referenced this section for further discussion on the serialization method. We are not going to start a pro or contra discussion about using XML. The fact of the matter is, that any kind of data serialization method – including XML – could be used, that can serialize data with lists hierarchical data – be it through a tree in XML or with a mapping method as done in JSON.

We have, in full view of the consequences, not defined a specific serialization method. Keeping the question of the serialization method open allows for more flexibility. It allows for any suitable serialization method to be used, including multiple ones. One could even imagine, without violating the requirement of staying platform independent, offering a *binary* serialization method – if a platform independent serialization method is offered as well.

7.3 Partitioning workflows across multiple WfMS

We originally planned to include the ability to partition process models onto multiple WfMS servers. In essence, every process fragment could be checked out, similar to activities, and returned after the execution of it completed. Instead of start parameters, the data containers of the context of the process fragment in question are supplied. However, since parallel branches may require synchronization, and exception handling may abort the checked out branch, extensive communication between the components may be necessary.

After reviewing [Ba01] we came to the conclusion, that although the idea is forth a discussion, further research is necessary. A fast and scalable implementation would require extensions on the meta models we are basing the WfQL onto: Synchronization edges may need to be specified in greater detail. Exception handling would need to be adapted and staff assignment may have to be evaluated before checking out partitions – or be implemented entirely different.

Clearly, more research is required on this topic that we cannot accomplish, as the main matter of this work focuses on a language's definition, and not fundamental workflow research.

8 Summary and outlook

In the introduction, we outlined general problems with current WfMS implementations. We isolated the problems to being limited to the interface of WfMS. To overcome these problems, we researched the requirements necessary onto such a WfMS interface in chapter 2. It became clear, that no existing standard covers all aspects and that a combination of standards would create conflicts due to the different meta models they are based on. For example, the Business Process Query Language [MS04] cannot be used in conjunction with the Web Services Business Process Execution Language [Me08], due to their conflicting assumptions on the underlying execution of workflows.

While looking for standards that would satisfy our requirements, it became clear, that to solve the requirements, a number of assumptions onto the meta models of the underlying WfMS have to be made. Therefore, we focused on researching fitting meta models for WfMS: We required a meta model for process models, an organization structure representation meta model for permission management and staff assignment and several more, that are discussed in chapter 3.

The meta models we chose implied a WfMS meta model which we defined in Chapter 4.

The language definition in chapter 5 is a direct result of using the previous research and combining it into a human readable, and computer parseable language specification. It is based on the WfMS meta model from Chapter 4, the general meta models from Chapter 3 and the requirements from Chapter 2.

Following the language definition, we wrote the prototypical server implementation and the client applications. At this point, it became clear that we could not find an existing WfMS that could satisfy several key assumptions on the meta model, such as data typification, an Adept compatible process meta model and monitoring capabilities on activities and processes. Therefore, we decided to implement a WfMS from scratch. This decision, although without viable alternative, has been impending us from presenting a greater bandwidth of

supported operations due to the amount of work necessary to write a WfMS – opposed to our original plan, of writing a WfQL server interface for an existing WfMS.

However, we have implemented several key functions, that have not yet been seen in other systems and are confident, that the language will be fully implemented in future works. We also hope, that the aspects discusses in chapter 7 are going to contribute to future enhancements of the WfQL.

List of Figures

Bibliography

[Ba01] *Baur, T.*, **Effiziente Realisierung unternehmensweiter Workflow-Management-Systeme**, PhD thesis, *Ulm University*, 2001

[Be05] *Berroth, M.*, **Konzeption und Entwurf einer Komponente für Organisationsmodelle**, diploma thesis, *Ulm University*, 2005

[BPMN] OMG, **Business Process Modeling and Notation V2.0**, 2010

[CORBA] OMG, **Common Object Request Broker Architecture (CORBA/IIOP) Specifications 3.1**, 2008

[Fo09] *Forschner, A.*, **Fortschrittliche Datenflusskonzepte für flexible Prozessmodelle**, diploma thesis, *Ulm University*, 2009

[He06] ACM, *M. Henning*, **The rise and fall of CORBA**, 2006

[Hib] Hibernate, `http://www.hibernate.org/`

[IEEE 1003.1] **IEEE Std 1003.1**, 2004

[ISO 8879] **ISO 8879**, 1986

[ISO 14977] **ISO/IEC 14977 : 1996(E)**

[Ju06] *Jurisch, M.*, **Konzeption eines Rahmenwerkes zur Erstellung und Modifikation von Prozessvorlagen und -instanzen**, diploma thesis, *Ulm University*, 2006

[JSON] **RFC 4627**, 2006

[Li+88] *Linnemann, V. et al*, **Design and implementation of an extensible database management system supporting user defined data types and functions**, Proceedings of the 14th International Conference on Very Large Data Bases, 1988

[LISP] *John Allen*, **Anatomy of Lisp**, Mcgraw-Hill College, 1978

[Me08] *Mehliß,S.*, **Verwendung von BPEL zur Service-Orchestrierung innerhalb einer J2EE-Umgebung**, diploma thesis, *FH Braunschweig/Wolfenbüttel*, 2008

[MS04] *Momotko, M. and Subieta, K.*, **Process Query Language: A Way to Make Workflow Processes More Flexible**, Proceeding of the 8th East European Conference on Advances in Databases and Information Systems (ADBIS), 2004

[POSIX-1003.1e] **POSIX-1003.1e**, 1999

[Re00] *Reichert, M.*, **Dynamische Ablaufänderungen in Workflow- Management-Systemen**, PhD thesis, *Ulm University*, 2000

[Ri04] *Rindele, S.*, **Schema Evolution in Process Management Systems**, PhD thesis, *Ulm University*, 2004

[Spring] Spring Framework, `http://www.springsource.org/about`

[SQL] **SQL:2008 ISO/IEC 9075**, 2008

[WfMFS] OMG, **Workflow Management Facility Specifications, V1.2**, 2000

[WRS08] *B. Weber and M. Reichert and S. Rinderle-Ma*, **Change patterns and change support features-enhancing flexibility in process-aware information systems**, Data & Knowledge Engineering archive Volume 66 Issue 3, 2008

[XPDL] OMG, **XML Process Definition Language 2.1**, 2008

Name: Mark Oliver Schmitt Matrikelnummer: 519573

Erklärung

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

Ulm, den .....

Mark Oliver Schmitt