UNIVERSITY OF ULM
Faculty of Computer Science
Department of Database and Information Systems

Emergent Workflow

Diplomarbeit
presented by

Florian Bertele

Thesis advisers: Prof. Dr. Manfred Reichert
Prof. Dr. Peter Dadam

April 2005

Abstract

Many fields of work require information systems that support an organization in managing its complex process-aligned business. However, the flexibility of process creation and enactment offered by current workflow management systems is often insufficient. As a consequence these systems are not broadly used and suffer from low acceptance. Agile processes that involve creative work are hardly supported as requirements changes and exceptional situations occur frequently. Emergent Workflow is an approach that tries to overcome these deficiencies by capturing the current process instantly – as it *emerges* out of work – and offering immediate support to workflow participants. Its goals are the retainment of organizational knowledge, improved reuse of individual work patterns and a better transparency of the overall process. This thesis first motivates the subject by introducing a field of application in automotive product development. Typical components of an Emergent Workflow Management System are identified and their requirements as well as a process model are specified. Then related work is presented and matched against these requirements. The thesis closes with a conceptual architectural proposal and discusses some issues of feature integration and implementation.

Contents

1. Introduction

1.1. Motivation

Today, entrepreneurial success is determined by both external and internal factors. As the economic competition grows harder, companies face several external challenges. The high innovation speed in research and production leads to shorter product life cycles and less development time. Markets tend towards going global and offer more choice for customers. Thus customers' expectations towards competitive pricing, quality, performance and flexibility of products rises as well. Internally, production and development gets more and more complex with each generation. To handle that complexity, staff becomes highly diverse and develops specific knowledge in each department. That makes it harder to aggregate each individual's work and to communicate common goals. Obviously, complicated products cause complicated corporate structures. That is why companies have come to extend their focus from a product-oriented view to a more process-oriented view.

By aligning business in a *process-oriented* manner, inputs, outputs and relationships between activities have to be identified. Formalizing these elements helps to break down the corporate strategy into operations and clarifies their relation. The process itself of creating explicit process models and visualizations fosters a more in-depth understanding of collaboration and the flow of documents, products and work. A more transparent perception helps to spot chances to increase efficiency such as eliminating redundant work, defective products or reducing cycle times. According to Jablonski and Bussler [JB96], the expected benefits are among others improved quality of service, improved productivity and cost reduction and reduced vulnerability of the work process.

Workflow management systems have been introduced in order to give technological support to the idea of business processes. They are software systems dedicated to manage the steps involved when dealing with business processes, such as modeling or assigning tasks. A workflow management system is meant to encapsulate all process logic within a corporate information technology system.

The classical model of a *business process life cycle* is depicted in Figure 1.1. *Process design* is the task of distilling a process model from a set of informal business requirements. It involves the definition and selection of appropriate tasks (possibly from a task library), sequencing of the tasks to satisfy data and logical dependencies, allocation of resources consumed by tasks, allocation of agents to execute tasks, scheduling of tasks considering concurrency, and finally validation and verification of the model.

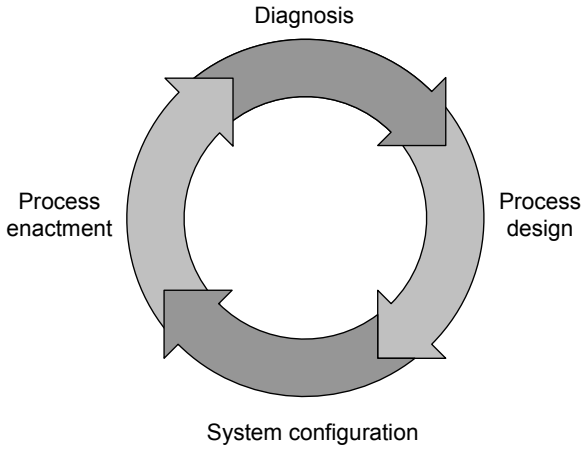


Figure 1.1.: The business process life cycle (compare [Aal02] Figure 2)

During *system configuration*, an initial business process is implemented and deployed in the workflow management system. In the following *enactment phase* instances of the implemented models are created and executed. A process instance passes a number of states by initiating its tasks. The conditions and sequence of task execution is stated in the process schema as well as a terminal state. After enactment, the process instance history is *diagnosed* for analysis and improvement. Conclusions drawn from that phase influence the next *process (re)design phase*.

However, workflow management systems have not been accepted widely in pracice been [HSW97]. Multiple reasons can be found for that: Technology sometimes has not been proven to be mature enough for corporate-wide deployment. On a managerial level people may be not convinced of the positive effects a workflow management system has on efficiency and see primarily high investments. As most activities of employees can be individually controlled and monitored by information systems, acceptance problems become apparent as well. People feel observed or are afraid of doing "office assembly-line work" due to the high degree of work assignment automatization. The most profound deficiency though is the lack of flexibility in most commonly used workflow management systems.

Depending on the type of work and operational business, it is quite common that, from time to time, the product or the production process needs to be changed. Exceptional situations occur that have to be treated separately. These might be caused due to internal or external events, such as special arrangements with a customer or extra quality checks due to legislative changes. Very often it is not possible to foresee all possible exceptions during process design, so the implemented workflow model does not cover it. What happens most of the time is a treatment of such cases out of the system. Activities are inserted, modified or skipped manually without proper documentation – the workflow management system does not know anything about the deviation from its standard procedures. Such behavior leads into a situation where processes (or what is

left of them) become intransparent and the knowledge about them is incomplete or even incorrect. Since this would practically reverse all efforts put into process management, exceptional situations need to be taken care of differently.

The correct resolution of an incomplete process implementation is another reiteration through the business process cycle (see Figure 1.1): Let process designers and staff diagnose the weaknesses, redesign their models and get them implemented into the workflow management system. Furthermore running process instances need to be taken care of separately to assure their conformance with the new model. That is a tedious and time-consuming task that involves many reports, meetings and interviews. Design work and communication between various groups of people leads to a certain degree of information loss and potential misunderstanding. If such changes appear very frequently on potentially long-running process instances, the efficiency advantages of workflow management system are mostly lost. Due to these shortcomings of conventional workflow management systems, we motivate the use of flexible workflow management.

A *flexible workflow management system* is able to adapt to changing requirements of its users and their work items, particularly during process enactment. That includes the consideration of exceptional situations, ad-hoc changes to workflow instances, activities and resources and workflow schema alteration. Knowing that not even the most carefully pre-built process model suits all possible future situations and later alterations are unavoidable, a flexible workflow management system rather focuses on offering means to extend or modify its behavior for all involved parties in an acceptable way. It does not force users to circumvent its limited capabilities outside the system, but lets them document a change operation and its context as well as possible.

1.2. Vision of Emergent Workflow

Emergent Workflow envisions a flexible workflow management system with the capability of building small-scale workflows during process enactment without explicit process design.

There exist many fields of work which share characteristics such as being highly variable and having low regularity patterns in their schema of activities. For instance, highly creative or knowledge intensive processes like product development fall into that category. At the same time, those kind of processes require close collaboration of many people from several disciplines, each representing distinctive knowledge. There exist many different views on one common project, all of which need to be integrated properly.

Introducing a workflow management system into an environment like that is very promising due to the large amounts of implicit knowledge involved. Building an information system that collects structured information about the process and makes it available for later reuse would yield the benefit of improving each individual's process awareness and productivity. A higher work pace, work quality and learning curve are among the

potential benefits.

However due to the nature of creative work, a small scale process can hardly be pre-modeled because there does not exist literally one single regular case of reasonable complexity. Rather, there is a rough framework whose detailed structure is subject to continuous adaption due to spontaneous requirement changes.

As the exact process logic is unknown until process enactment, it is the approach of Emergent Workflow to capture the process as it *emerges* from spontaneous performance of activities. An explicit modeling approach is impractical as it is both too complicated and too time-consuming to be done by people who are not dedicated process designers. The user rather documents their advancement in a more convenient and less formal way, e.g. supported by a dialogue-based software. A partial process model is then supposed to be derived from an audit trail that documents users' activities.

Interesting uses for that information include documentation, reuse and composition. As for documentation, recurring situations including their context and decisions made upon them can be reviewed to gain insights for future work. If a very similar situation occurs in the future, it is even possible to reuse a previously recorded situation as a template to guideline upcoming activities. Finally, the collected set of small-scale process parts contains all information necessary to compose a view on the overall current process. This is particularly interesting to compare with a theoretically developed target process in order to find characteristic differences and chances for improvements.

1.3. Application example

In order to get a taste of what a typical application environment could look like, an initial example is introduced. It helps understanding the major questions that have to be asked and answered when considering the introduction of Emergent Workflow. Furthermore the application scenario is used throughout the thesis to illustrate proposed ideas.

The example introduces an outline of a *new product development process* in the automotive engineering sector. Automotive development is a relevant application field for Emergent Workflow for a number of reasons. Modern automobiles are *mechatronic systems* – machines whose components comprise mechanical, electronical and information technology aspects. Their correlation is visualized in Figure 1.2 and the meaning of mechatronics is defined by VDI [Ver04] as follows:

> *[**Mechatronics** is]...the synergetic integration of mechanical engineering with electronic and intelligent computer control in the design and manufacturing of industrial products and processes.*

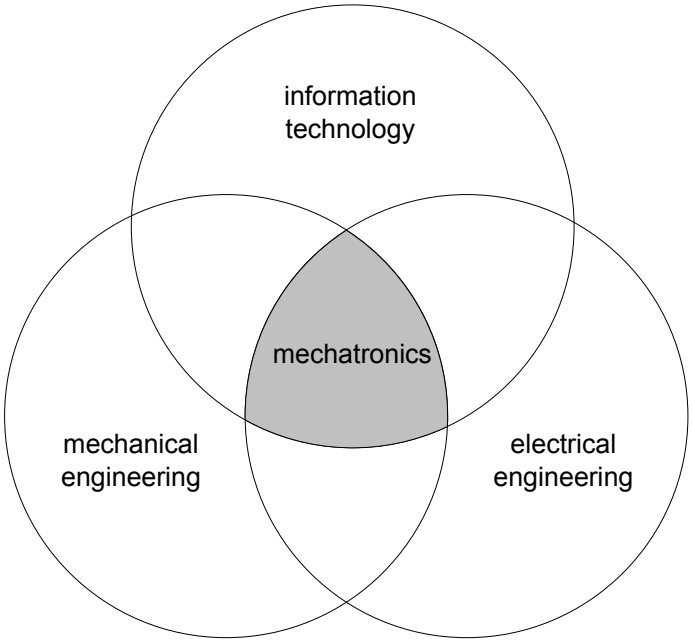


Figure 1.2.: Mechatronics as the interaction of different disciplines (compare [Ver04] Figure 2-1)

It is crucial to notice that *synergetic effects* can not be reached by independently operating science groups but take their power from cooperation with each other. That implies consequent synchronization between the disciplines to establish a common view, language and understanding of development issues.

The driving force behind interests in mechatronic systems is the fast paced innovative potential in information technology. On the one hand it is due to the exponential advancements of processing power and memory with concurrently decreasing costs and size at the same time. On the other hand the functional and spatial integration of technologies unleashes potential improvements concerning functionality, absolute performance and price-performance ratio as well as better behavior.

In an automobile, electronic and information processing components are built on top of a mechanical structure. This structure would suggest a sequential development procedure which is not practical in reality though because of its very time-consuming nature. For efficiency reasons it is rather desirable to have a continuous, distributed development and cross-domain cooperation at the same time. A *digital mock-up* is a widely used tool in product development to achieve that objective. It is a virtual prototype used by all involved disciplines to simulate and test the most important physical and other functional aspects.

As development procedures can not be pinned down to one single best model, the combination of the following patterns offers more flexibility:

- General problem-solving as a micro cycle
- V model styled macro cycle

Problem-solving as a micro cycle The problem-solving cycle shown in Figure 1.3 applies to small-scale procedures and comprises several components. The starting point is either a *situation analysis* or the *adoption of a goal*, depending on whether a pre-existing structure is adopted or new structures are built from scratch. After the situation has been analyzed with a given structure, a goal can be formulated from given input. In case an ideal concept is the starting point, these goals are adopted first and situation analysis starts from there.

During *analysis* and *synthesis*, a solution for the given problem is researched. Both activities analysis and synthesis are alternating: The first develops solution alternatives which are then checked, improved or rejected during synthesis of the results. By iterating these steps, improved solutions are eventually found.

The final *analysis and assessment* step evaluates the solution alternatives found in more detail. An assessment with regard to the initial goal formulation leads to either a proposal or a recommendation for one or more proposed solutions.

During *decision*, one compares the overall success of procedures which have shown satisfactory result so far. It either ends in a return to another goal formation if the results were not convincing or a favorite solution is chosen.

Planning for further procedure or *learning* makes sure that, at the end of one micro-cycle, the efforts made so far are carefully reviewed and evaluated. Learning about the good and bad points from the past cycle helps to improve further planning. That leads to a systematic improvement of future processes.

V model The V model is a macro-cycle that formulates – in contrast to a micro-cycle – a view on the overall development process. In Figure 1.4 multiple iterations of macro-cycles are shown.

The process starts at the entry point of the innermost cycle on its left side. From there, each iteration begins with a formulation of its respective *requirements*. They specify the goals of the macro-cycle in detail and are used as a comparative measure for outcomes.

During *system design*, developers establish cross-domain concepts for solutions. That is achieved by decomposition of major system functionalities, finding solution elements and recomposing these into an overall solution concept.

The *domain-specific design* phase is used by each discipline individually to elaborate on solutions that had been outlined during *system design*. Solutions are substantiated in more detail which requires separate models and views for mechanical, electrical and information engineering each.

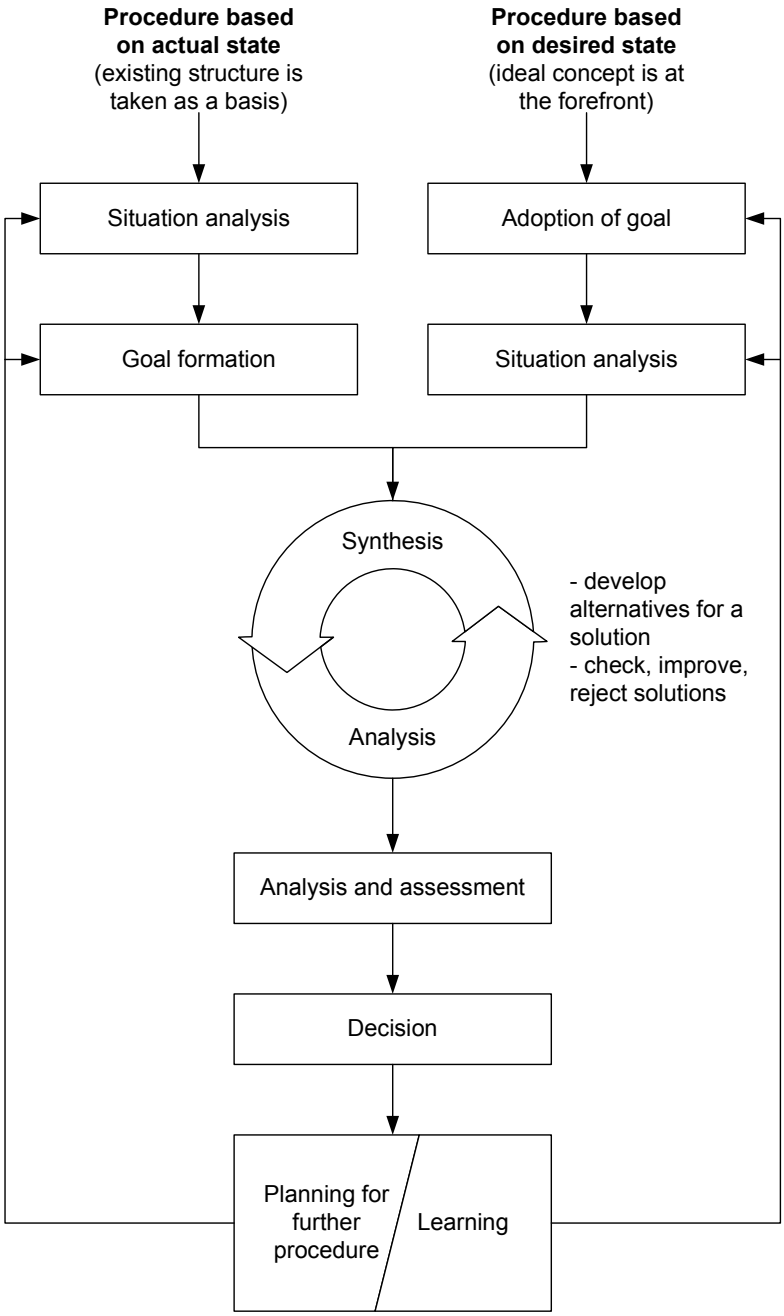


Figure 1.3.: Problem-solving as a micro-cycle (compare [Ver04] Figure 3-1)

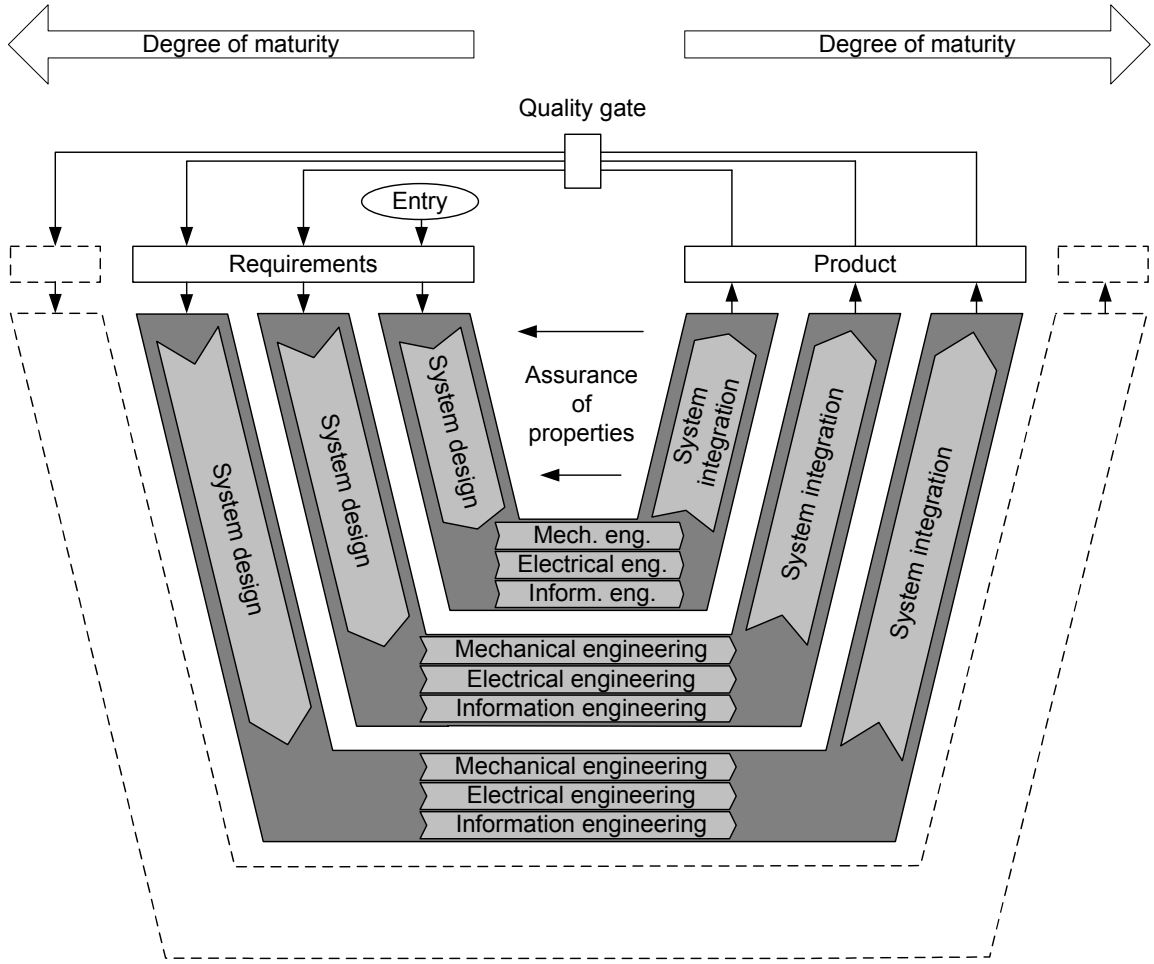


Figure 1.4.: V-model styled macro cycles with increasing product maturity (compare [Ver04] Figures 3-2 and 3-3)

System integration finally consolidates all partial solutions and investigates their interaction. An important part of integration is the *assurance of properties* as indicated in Figure 1.4 by arrows pointing from right to left. As integration proceeds, its results are continuously checked back with the solution concepts built during *system design*. Furthermore, their compliance with the initial requirements has be assured.

A macro-cycle iteration results in a *product*. This can be either the final product which is ready to be released or just an intermediate product such as a certain prototype stage. As complex products require usually several macro-cycles for development, an intermediate *product* has to pass a *quality gate* to proceed to the next development cycle. A *quality gate driven process* ensures the quality level of outcomes at a certain stage by defining detailed requirements that have to be met before entering the next stage. After passing the quality gate, the next set of *requirements* gives the agenda for the next macro-cycle and the next quality gate. With each additional macro-cycle, the *product maturity* in terms of completeness and correctness increases until the last cycle outputs the final product.

The *V model* and the *problem solving micro-cycle* indicate that Emergent Workflow is a promising approach in automotive development. On the one hand, the overall process has a coarse fixed structured defined by iterating through *quality gates*. On the other hand, small-scale problem solving appears frequently, is individually determined by the context and has little repetitive structure. Still, there are expected benefits from reusing previously applied procedures. Suppose a construction detail such as an advanced window power lifter. It may have already been implemented in a premium class model line successfully and is about to be adopted for the next generation of a compact car. The processes and insights recorded while integrating the power lifter in the premium car can save efforts by being reused for its integration in the compact car.

1.4. Terminology

While talking about a specific field of application, we have used a lot of terms without exactly specifying their meaning. This sections purpose is to introduce the terminology that will be used most commonly throughout the subsequent chapters. The following definitions and explanations were established by the Workflow Management Coalition in "Terminology & Glossary" [Wor99] respectively taken from [AH02, WRWR05]. We will adapt their interpretation in the following paragraphs.

A *business process* is a set of procedures and activities, which collectively realize a *business objective* such as the construction of a new car generation. These procedures and activities are linked by various relations, e.g. temporal or causal dependencies.

A *workflow* is the automation of a business process, in whole or partially. A set of procedural rules manage the exchange and distribution of documents, information or tasks. Strictly speaking, the term workflow refers to the subset of processes which are sup-

ported by *information technology*. Since however the differentiation of a process versus a workflow is not crucial in the light of this thesis, I will mostly use both terminologies synonymously.

The execution of workflows is defined, created and managed by a *workflow management system*. By the use of software it runs on one or more *workflow engines*. These are able to interpret formal *process definitions*, interact with *workflow participants* (also called *workflow users*) and, where required, invoke the use of applications and other information technological tools. A workflow participant is a human or machine-based agent that constitutes a resource which performs work represented by a workflow activity instance. A workflow management system that meets the requirements discussed in Chapter 2 will be referred to as an *Emergent Workflow Management System*.

The automation of a workflow is defined within a *process definition*. It is the representation of a business process in a form which is supported for automated manipulation, such as modeling or enactment by a workflow management system. A process definition holds a certain *process type*. The type is specified by a *process schema* which defines the process structure. The schema consists of a network of tasks and their relationships, constraints to indicate the start and termination of the process and information about individual activities, such as participants, associated applications and data, etc.

A business process is structured by the identification of logical steps. Each atomic step is referred to as a *task*. A task is performed by the execution of an instance-specific *activity*. During execution, an activity passes a sequence of defined states. Activity state traversal can be either workflow automated or manual without information technology support. A workflow activity requires human and/or machine resources to support workflow execution. Where human resources are required, an activity is allocated to a workflow participant.

A *process instance* is a process definition with individually allocated resources and activity states for all tasks it contains. The set of activity states defines the execution state of a process instance. During *process enactment*, a process definition is both instantiated and executed. That is, a process definition with an individual process state and its resources are allocated and an initial state transition is performed in order to indicate the instance's readiness. In literature, process instances are often referred to as *cases*[1]. In this thesis, we will stick to the term process instance in order not to confuse it with the term case used in case-based reasoning (see Section 3.1).

Many individual *process instances* may be operational during process enactment. Each process instance is the representation of one single enactment of a process definition and may be controlled independently. It has its own internal state and externally visible entity. A workflow management system creates and manages a process instance for each separate invocation of the process definition.

A *worklist* is a list of *work items* which are associated with a given workflow participant.

[1] e.g. by van der Aalst et al. in [AH02]

Each work item is a representation of a task which has been scheduled for execution in the context of an activity within a process instance. The worklist represents a part of the interface between a workflow engine and the *worklist handler*, a software component that manages the interaction between the user and the worklist. It enables work items to be passed from the workflow management system to users and forwards notifications of completion or other work status conditions.

1.5. Organization of this thesis

Up to now, **Chapter one** has motivated and introduced the subject around Emergent Workflow. After a *motivation* of business process management in conjunction with workflow management, the *vision* of Emergent Workflow is presented. A characterization of a possible field of *application* follows. The Chapter closes with an introduction and clarification of the *terminology* most commonly used throughout the thesis. **Chapter two** presents the requirements on an Emergent Workflow Management System in a structured manner. First, typical *use cases* are identifed, from which a *component overview* is concluded. Then detailed *requirements on each individual component* are elaborated. A tabular *requirements summary* gives a brief statement on the most noticeable points of the component's requirements. The Chapter closes with a specification of characteristics for a suitable process metamodel. **Chapter three** presents related work approaches to Emergent Workflow. *Case-basd reasoning*, *process mining* and *flexibility approaches* are introduced and assessed with respect to their usefulness for Emergent Workflow. **Chapter four** contains an *architectural proposal* for an Emergent Workflow Management System presented in *three stages*. **Chapter five** discusses *functional issues of integration*. Finally, **Chapter six** contains a *summary* of the presented work, a *conclusion* and mentions *future and omitted work*.

2. Requirements

In order to receive functionalities as described in the vision of Emergent Workflow, certain requirements have to be met. This Chapter attempts to explore these and lay out some details about them. First, an overview of typical use cases identifies user groups and their interaction with system components. That information is used as a starting point for a more complete illustration of all generic components of Emergent Workflow. From there, each mentioned component is further elaborated concerning its interfaces, functionalities and constraints. After a summary of component-oriented requirements, requirements on an underlying process metamodel follow.

2.1. Use cases

Process design Although Emergent Workflow aims at a more spontaneous creation of process models, pre-modeled processes can not be left out in practice: On the one hand, they may be still used as a starting point for process development and on the other hand a coarse, big scale process model can be used for flexible workflows as well.

Example 1. The *V model* depicted in Figure 1.4 shows the common procedures for automotive development. Although it is a highly creative process, there is a rigid framework of steps to take during development: There is a number of quality gates to pass, each with a dedicated design phase, discipline-specific problem solving and a final integration phase.

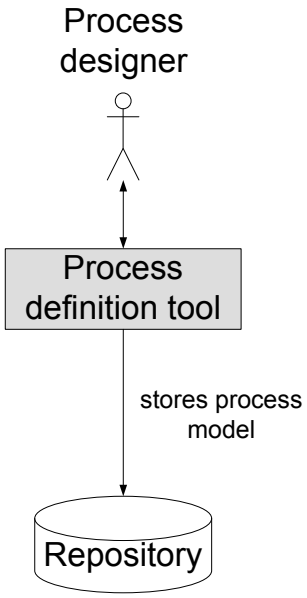


Figure 2.1.: Use case: process design

Before enactment, a dedicated process designer models explicitly a more or less complete process model. It consists of the overall structure determined by a development model and also generic procedures which have standardized and repetitive character. This model is being formalized by the help of a process definition tool and stored to the repository.

Administration A process model is the starting point for enactment of process instances. Instances may be initialized by users or an administrator using a *user interface* to the *runtime engine* of the workflow management system. After the instance is up and running, it is being managed by the administrator until it reaches a terminal state. Management includes observing functions such as monitoring the progress and state of instances, intervening in exceptional cases and overriding user interactions as necessary.

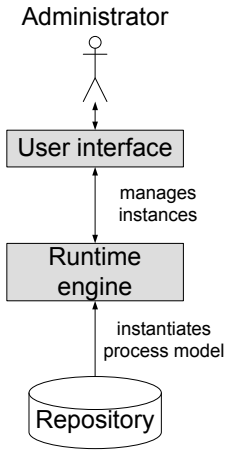


Figure 2.2.: Use case: administration

Usage & creation A workflow user is someone whose work is coordinated by a workflow management system. In Emergent Workflow, this person (or agent) does not only receive tasks from the runtime engine, but is also involved in creation and adaption of partial processes. This is possible and necessary as the flexible approach of Emergent Workflow intends to give its users the freedom for self-determining, thus creating their own partial process. So after the reception of a task through a *user interface* or a *client application*, the participant performs the steps necessary to complete the task. His actions are being formalized in an *interaction protocol*. This protocol contains the information which is necessary to reconstruct the user's individual process fragment in a *process creation engine*. Finally, this fragment of a process instance is stored to the *repository*.

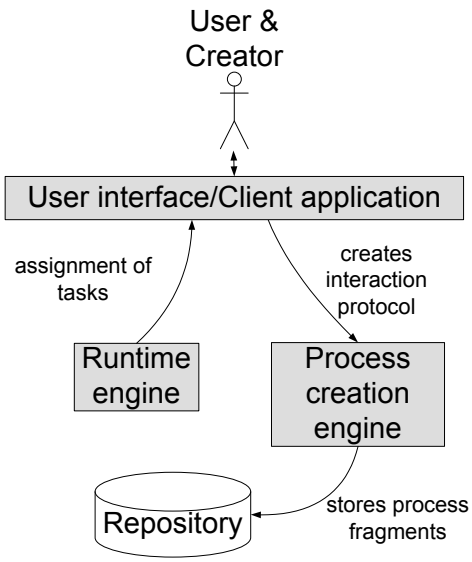


Figure 2.3.: Use case: usage & creation

Composition Having stored these process fragments, the process designer can now go ahead and compose these elements into bigger compositions. These foster the understanding of the coherence of collaborative work and can be used either for documentation or as a template for big-scale process redesign. As there are probably many fragments available in the *repository*, a designer needs the support of a *process matching engine* component which assists him finding relevant fragments. These are composed in a *process modeling tool* and resulting compositions are stored to the *repository*.

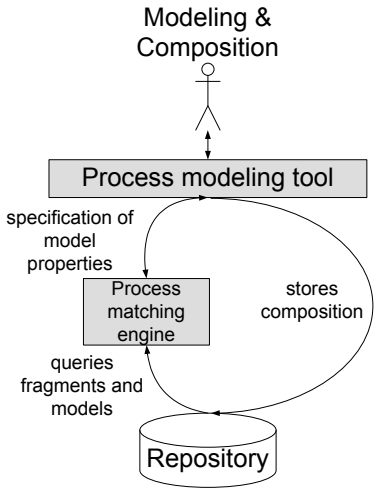


Figure 2.4.: Use case: composition

Usage & reuse Once the *repository* is filled with process fragments, a workflow user may now choose to make use of them. So when the *runtime engine* assigns him with a task that turns out to be similar to a task which has been processed in the past, the user may choose to follow similar procedures again. Thus, he will rely on the *process matching engine* to find a template in the form of a stored process fragment. That template guides him at a chosen level of interactivity through the procedures. As it is in creative processes likely that spontaneously formed processes slightly differ from each other, deviations from the templates occur and are recorded again in an *interaction protocol*. As before, the trail is transformed by the *process creation engine* into a new fragment and stored to the *repository*.

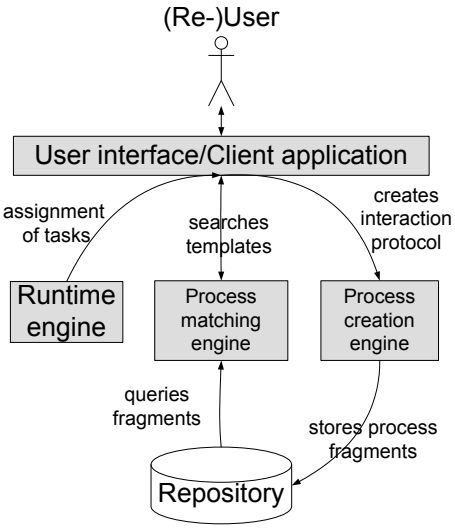


Figure 2.5.: Use case: (re-)usage

Documentation The last distinguished use case is the role of documenting work. Process fragments can be documented already at run time with annotations, however separate documentation may summarize the most important insights from a post-hoc point of view. These records may require references to process fragments as they were derived during the execution of the current process. Again, a *process matching engine* is needed to spot the relevant fragments and integrate them on the client side with a *documentation tool* and store the results back to the *repository*.

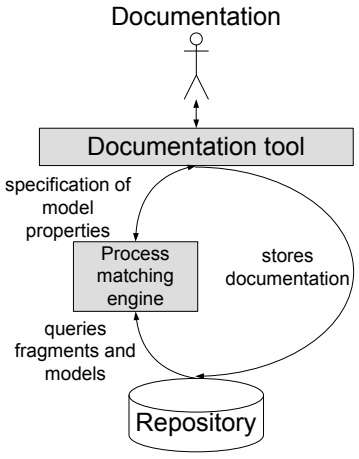


Figure 2.6.: Use case: documentation

2.2. Component overview

While enumerating use cases, basic components of Emergent Workflow were mentioned. In order to receive a more complete understanding of all the components involved, this section gives a short overview of them. As the usefulness of a workflow management system is not only determined by its functions but also by its ability to interact with external entities, a set of standardized interfaces has been defined by the Workflow Management Coalition. Their Reference Model [Hol95] is shown in Figure 2.7.

The model generalizes the idea of a runtime engine to a *workflow enactment service*, as such could potentially contain multiple workflow engines. This service is encapsulated by a Workflow API and interchange formats which are the results of standardization efforts by the Workflow Managements Coalition.

It distinguishes five interfaces in total:

Interface 1: Process definition tools This is the interface used during process design phase by process designers to transfer developed process models to the workflow management system.

Interface 2: Workflow client applications All workflow-related user interaction is directed over this interface. Typically this includes client applications that manage assigned work items for users and updates the system about work progress.

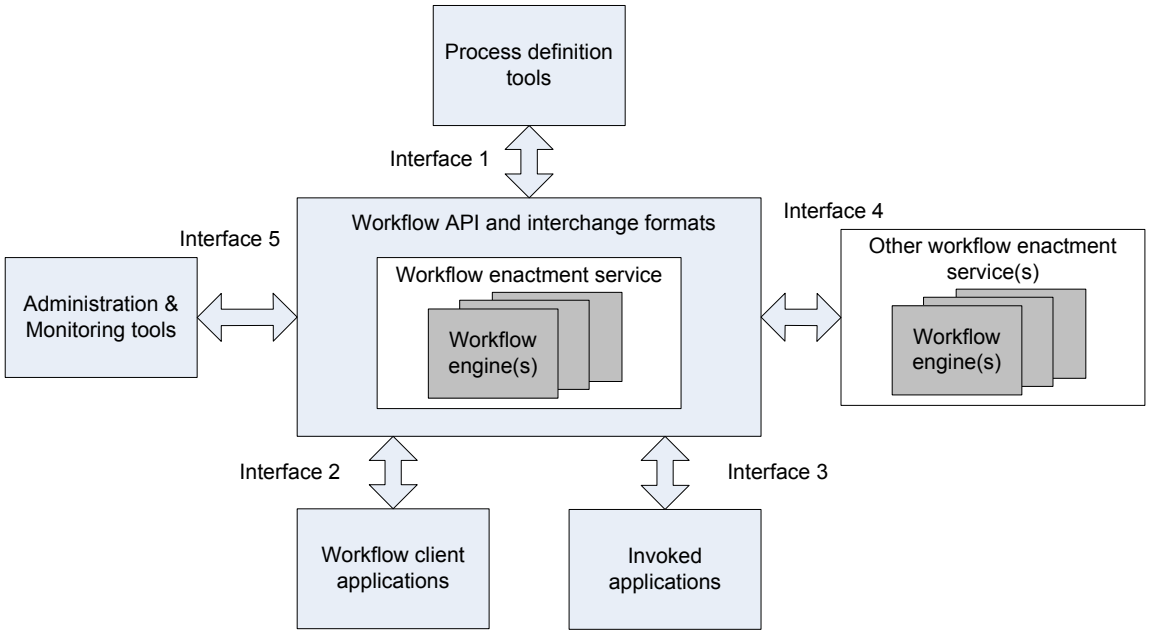


Figure 2.7.: Workflow Reference Model (compare [Hol95] Figure 6)

Interface 3: Invoked application This interface addresses third-party applications which are invoked server-side by the workflow enactment service such as Enterprise Resource Planning software.

Interface 4: Other workflow enactment service(s) Cross-organizational workflow becomes a hot issue when a combination of services offers additional benefits. This interface serves the purpose of enabling interoperability between various types of workflow management systems. They exchange use and control data, enable synchronization and virtually merge independently created and executed processes.

Interface 5: Administration & monitoring tools Administration and monitoring is a default requirement for any workflow management system. Therefore, a generic interface is defined which allows the use of non legacy applications for administration and monitoring.

Figure 2.8 gives an overview of all identified Emergent Workflow components. Three groups of components were identified and aligned in an interface, logic and data layer each. *Interface* components direct and format relevant input or output data. Components for the application *logic* process data and forward outputs to the other two layers. The *data* layer finally handles storage of data.

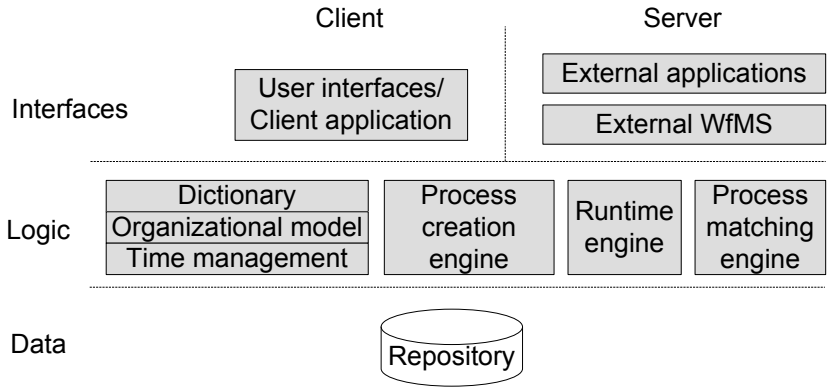


Figure 2.8.: Emergent Workflow components

2.3. Component-based requirements

In the following Sections, the desired functionality of all mentioned components is explained and functional as well as nonfunctional requirements are derived.

2.3.1. User interfaces/Client application

A user interface represents all users' access point to the Emergent Workflow Management System. Notice that we summarize interfaces 1, 2 and 5 from Figure 2.7 into this generic Section about user interfaces/client application. Hence three different user groups, workflow participants, designers and administrators apply varying functional and nonfunctional requirements on this interface. They have been combined into this section as a detailed specification of functional requirement of applications used by designers and administrators is not a point of emphasis in this thesis.

Functional requirements *Administrators* require applications that allow them to control the workflow management system with special focus on the runtime engine. Aspects such as instantiation, execution, termination of instances as well as their permanent placement in an archive are to be monitored and influenced as necessary. *Process designers* analyze, create and compose process models. Hence their client applications are to provide support when retrieving running or archived process instances and during the creation or composition of process models. *Human workflow participants* (from this point on also referred to as "the" users) require client applications that receive incoming work items representing tasks, manage this set of tasks using a worklist handler, help them document their work and return status information to the system, such as when the user has started or finished their job on one work item. Non-human workflow participants referred to as *agents* have special requirements regarding a machine-readable interface, but behave generally very similar to human users and are therefore not further

considered here.

A *user interface* is either referred to as a part of the *client application* or represents the client application itself, commonly depending on whether there is enough application logic present at the client: A tool that graphically lists all incoming work items is usually called an user interface, whereas a version of this tool supplemented with functionality for execution and manipulation is rather called an application. In both cases, their appearance is critical to the acceptance of the whole workflow management system. That is, nonfunctional aspects determine whether a software system is understood and controlled by users to its fullest extent or its features are mostly ignored and worked around.

Application integration levels describe the functional level on which client applications can access a workflow management system's functionality and vice versa. At a minimum integration level, the runtime engine may receive the ability to start/stop a client application upon the start/stop of an activity. In a second level, startup parameters can additionally be handed over to a client application which itself hands back a return value upon its termination. At the next level, the ability to pass data objects as input or output for the application may be added. The highest level of integration of a workflow management system and a client application represents a module or macro call type of access directly through the client's respectively the workflow management system's API (Application Programming Interface). The implemented level of integration determines to a certain extent the ability to automatize a process and thereby increase user efficiency.

Usability User interfaces and usability in general are a wide field of studies; this paragraph does not intend to claim completeness on this side aspect of the thesis. It is rather meant to provide a starting points and examples of objectives to consider. For a more elaborate discussion of usability, appropriate literature exists[1]

In order to help the novice or occasional user to make his first steps with workflow management, an easy to use interface is substantial. *Intuitivity* and *simplicity* are two very frequently mentioned nonfunctional requirements for any user interface. The first may be described as the ability of an interface to behave in all situations as expected by its typical user. Simplicity is a very delicate issue, as it runs contrary to most functional requirements: To give users a clear understanding of how they are supposed to interact and what their actions will infer. This goal is mostly reached by a low number of items on the screen and predefined screen sequences (such as "assistants" or "wizards") which makes it hard to integrate a lot of functions in the interface. The simpler the interface, the lower is the learning curve for its users to work at a high level of productivity.

Additionally, *documentation* is an important aspect in order to achieve acceptance for

[1] For example Dix, Finley, Abowd, Beale "Human-Computer Interaction" [DFAB98] or Shneiderman "Designing the User Interface" [Shn98]

a user interface. As a persistent and complete understanding of all aspects of a user interface is rather less likely for all potential users, proper documentation helps them to answer raising questions on their own.

Configuration & customization A system environment differs individually from client to client: Invocation of various third-party applications needs to be *configured* individually on each system. Also, once a user becomes more advanced in using an interface, he might want to modify its behavior in order to enhance his working speed. As there is not one uniform user, there does not exist one perfect interface that meets all users' needs as well. While an explanatory pop-up window is helpful for the novice user, it is annoying and useless for the advanced user. *Customization* describes those abilities of an interface, e.g. to modify its look-and-feel, toggle optional parameters, add keyboard shortcuts and adjust the level of interactivity.

Interaction protocol Apart from user communication, the most important functional requirement can be seen from the use cases in Section 2.1. A user interface has to propagate user interaction in the form of an *interaction protocol* back to the workflow management system. It is one of the key ideas of Emergent Workflow to derive complete or partial process or instance models from recordings of spontaneous flow of work. This can only happen if there exists sufficient input which has been generated on the client interface layer. An adequate interaction protocol contain the sequence of actions of a user including their context and modified data objects. The more complete and consistent user interaction can be formalized, the more it is likely to come up with correct conclusions regarding the current in-detail process.

2.3.2. Server interfaces

Although this thesis does not deal with server-side interfaces in-depth, for the sake of completeness they are mentioned here shortly. Two interfaces assure the integration of workflow management systems into an existing and heterogeneous environment: One for *external workflow management systems*, the other one for *external applications*.

Communication between workflow management systems is motivated by a trend towards closer collaboration between companies, such as along the value chain of a modular and complex product. The consequence is that companies using process aligned information technology start sharing certain portions of their internal process in order to improve collaboration. Cross-organizational workflows are an example for the alignment of multiple individual workflows into one virtual big workflow[2].

[2]Compare for example C. Bussler "The Role of B2B Protocols in Inter-Enterprise Process Execution" [Bus01] or Grefen et al. "CrossFlow: cross-organizational workflow management in dynamic virtual enterprises" [GAHL01]

For interoperability, an XML based protocol Wf-XML has been proposed for run time integration of process engines[3]. Different levels of interoperation are separated depending on the following scenarios: Cooperation may be chained where output items are passed on as input items for the next process. A nested subprocess can be found where a sub-task is performed by an external entity. A peer-to-peer organization describes independently acting entities that send work items as unsynchronized packets, whereas in contrast to that a parallel synchronized top process is established.

Non-workflow external application integration of workflow management is needed to put the abstract view of process instances into practice and execute them. Involved external applications may be as fundamental as a database system, an automotive production control system or as the classic example, enterprise resource planning software. There exist applications which are "workflow enabled" and those which are not; in the latter case an intermediate "Application agent" is used, otherwise communication may function directly. A standardized Workflow Application Programming Interfacen (WAPI) for synchronous/asynchronous access and data exchange has been established[4].

Analogue to the client interface, the creation of *interaction protocols* for all communication passing the server interfaces is a vital part for the functionality of Emergent Workflow. As user interaction is complemented by system reaction, both sides need to be recorded in order to draw a complete picture. Such systems do not only reside at the client side, but primarily at the server side as the examples given in the paragraph above illustrate.

2.3.3. Dictionary

When many users document their work progress, their input is used to build formal fragments of each individual's stake in the development process inside the workflow management system. As different users may enter the same data redundantly or use the same terminology in a different context, it is important to keep an eye on data consistency. Without an explanation and knowledge of the field of application, benefits from having the process documented are very limited. To avoid such ambiguities, it is suggested to establish a common syntax for all terminology which is used to describe work and its outcomes. Otherwise it is not possible for the system to grasp commonalities in related activities described by different users, if they use heterogeneous terminology for the same facts without specifying the semantic contents of their vocabulary.

Such confusion is avoided if all entered data is based on a previously or concurrently defined common dictionary. It defines shared terminology and highlights relations between terms like entities being synonyms, antonyms and homonyms. Authorized roles should be able to extend, modify and use this dictionary while documenting their work. A well developed dictionary is very valuable as it bears a formalization of various views

[3]See Wf-XML 2.0 Current Draft: http://www.wfmc.org/standards/docs/WfXML20-200410c.pdf
[4]See WAPI Version 2.0e Specification: http://www.wfmc.org/standards/docs/interface2-3.pdf

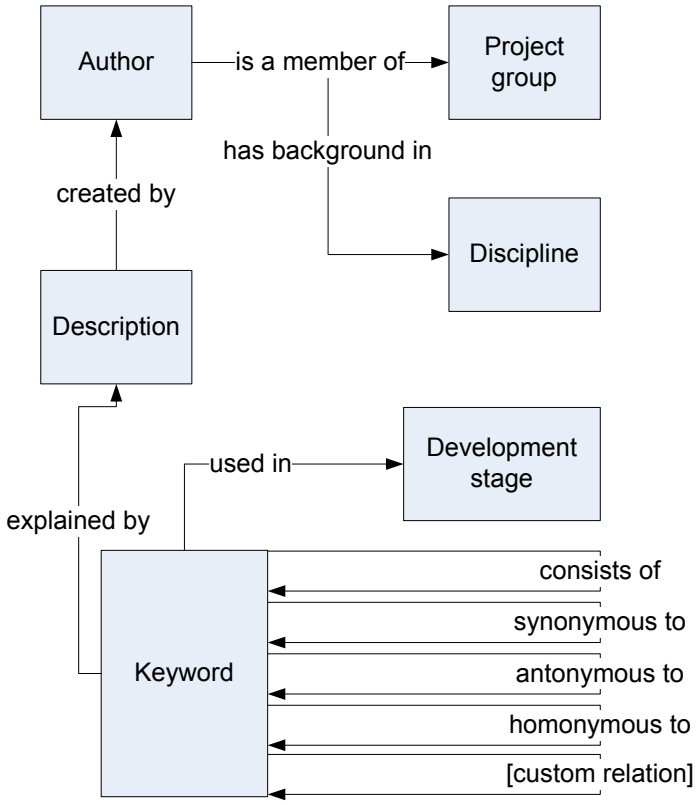


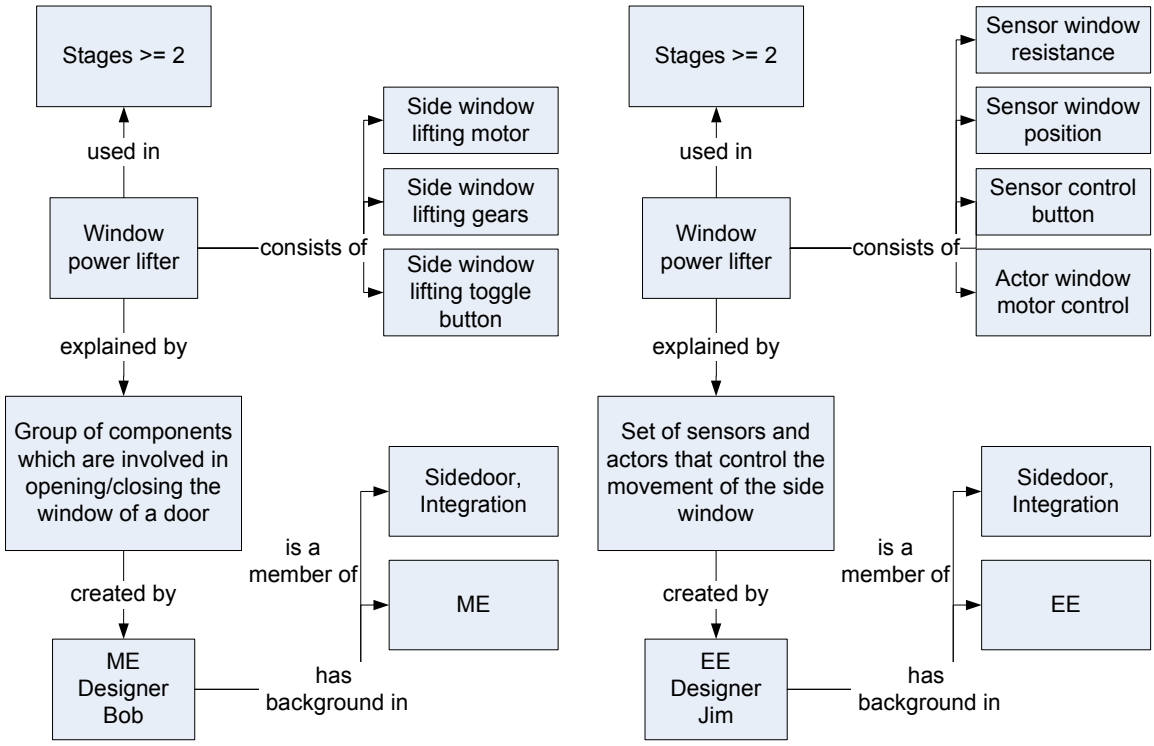
Figure 2.9.: Dictionary entities

and references on the subject which is being worked on.

Figure 2.9 shows a exemplary view on entities which are most likely to be chosen for a dictionary in an automotive new product development context. The core of a dictionary is the set of keywords it contains. During any rather complex process, it is very likely that a large number of keywords is being used and thus the dictionary grows quite big. In order to keep the dictionary still useful, it is essential to add supportive data in order to categorize its content. If the context of a keyword is stored additionally, it is easy to apply methods of data retrieval and modification just like in relational database systems.

The relevant context of a keyword is for example its description, which yields a textual explanation of the key term. As the same word can be used in several development stages with different meanings, one keyword can have multiple descriptions. Moreover, the author and his background regarding his discipline and role as well as the project group he is working in determines the usage and thereby the description of a keyword as well.

Furthermore, relations between keywords themselves should be expressed in a dictionary as well. Common relations are "consists of", which specifies hierarchical dependencies between keywords, "is synonymous to" , "is antonymous to" and "is homonymous to". Additionally, it is meaningful to allow process designers to create custom relationships



(a) Mechanical engineering view

(b) Electrical engineering view

Figure 2.10.: Examplary views of disciplines on the keyword "window power lifter"

within the dictionary, e.g. "is called by mechanical engineers ..." or "is named in the new development generation ...". Extensibility is crucial to the adaptability of a dictionary to changing requirements – consequently users will only make use of the dictionary if it supports their needs within their specific environment.

Example 2. Figure 2.10 shows an example of two different views on the component "window power lifter" within the automotive development process. In a mechanical context (Subfigure 2.10(a)), the window power lifter is regarded as an assembly group of gears, a motor and controls. An electrical engineer's view (Subfigure 2.10(b)) however focuses rather on the sensors and actors of that component.

This idea is closely related to the efforts being made in the Semantic Web movement. Its goal is to structure the contents of the World Wide Web in a way that allows both humans and machines to capture the semantics of the information available. The approach is to establish an ontology which is a conceptual schema that defines a data structure with entities, relationships and rules for a given domain.[5]

A dictionary as described defines a corporate-specific ontology that yields information about types of employees and their relations. That way, it is not only an information

[5]Compare http://en.wikipedia.org/wiki/Semantic_web

source to human users, but creates a machine-readable representation of domain-specific knowledge which builds the foundation for applications that support e.g. semantic composition of process fragments.

2.3.4. Organizational model

Employees perform different tasks according to their responsibilities within an organization. Consequently, a commonly used information system needs to adapt to each type of user by the provision of individually tailored support. That is why – apart from security reasons – authentication systems are gatekeepers to any kind of multi-user software using personalized applications or data.

A workflow management system additionally controls work activities and assigns work items to process participants. In order to abstract from individual users, sets of skills and responsibilities are subsumed to identify common *roles* within an organization. The execution of tasks is usually bound to a particular role, which means that the work item can be processed by any user holding a matching role.

Abstracting roles from individuals helps to distribute work load automatically as equally as possible within available personnel. Another benefit is the handling of exceptional situations like unavailability of a user. Dynamic rescheduling of work items to a work list of a substitute process participant makes it possible to avoid high variance in waiting time for work items.

When role abstraction is enriched with hierarchy information and roles are put into relations with each other, an *organizational model* is created. It represents the translation of a corporate personnel structure into an workflow model as seen from an organizational perspective (see also Section 2.4). Obviously that includes the hierarchic order and composition of organizational segments. Each individual has for example an educational background in a certain discipline, but can also have other responsibilities like executive tasks. So the fact that one person acts within several roles has to be formalized. Relationships like being subordinate or superordinate can exist between persons or only between certain roles of persons. Moreover, one person can participate with each role in different projects or task forces with overlapping responsibilities. Figure 2.11 illustrates these basic relations.

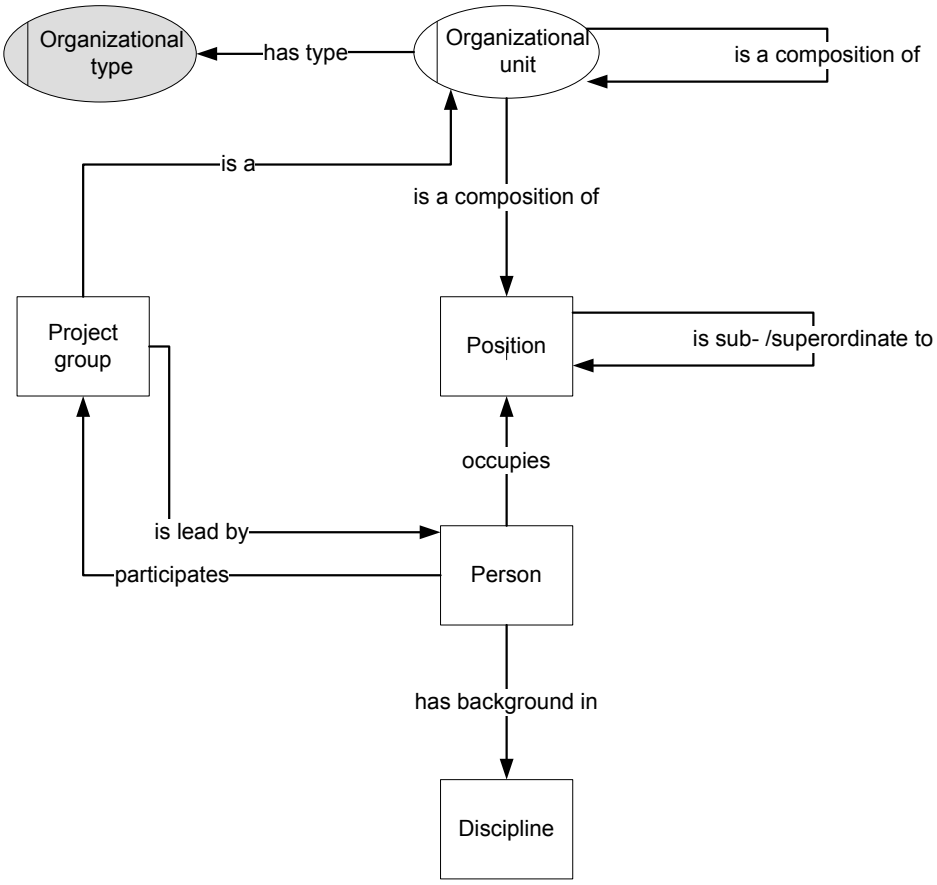


Figure 2.11.: Organizational Model

Example 3. An example of a basic organizational model is given in Figure 2.12. It refers to the running example of an automotive development environment. The automotive development unit has a type "development unit" and is lead by a head of development which is supported by an assistant. The unit splits up in three departments, each dedicated to the three disciplines involved in mechatronics (see also Figure 1.2): Mechanical engineering, electrical engineering and software development. Each department comprises a number of employees who perform one (or more) of the listed roles: A head of department with assistant, designers, engineers, quality assurance for testing purposes and people for documentation. That workforce is distributed over a number of project teams, where each individual gets assigned to projects according to his role. As an example, projects "chassis" and "sidedoor" are shown. The third project "integration" in the schema indicates that projects are not independent from each other. As component integration is a complicated task in automotive development, a dedicated project "integration" focuses just on integration issues.

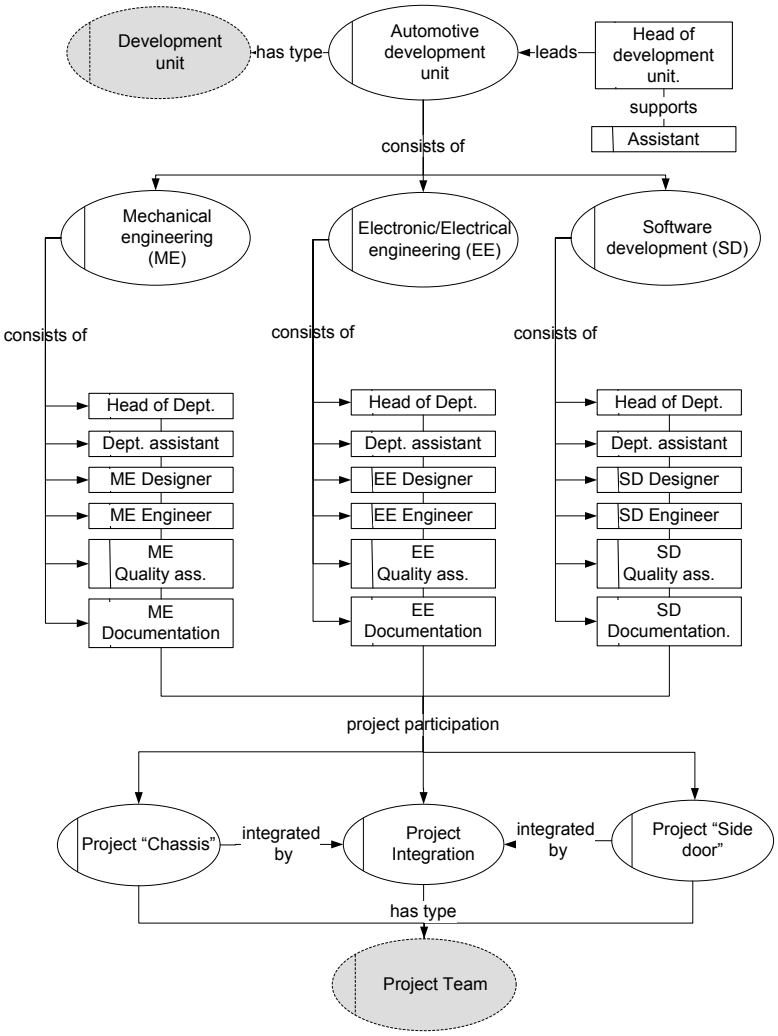


Figure 2.12.: Example: Organizational model

Creation of an organizational models starts with identification of existing personnel relations. Its usability is determined by its completeness and level of detail. Only roles that have been explicitly identified exist in an information system. In real-life organizations, employees hold official and unofficial roles representing their primary and secondary, often implicit tasks. On the one hand it is meaningful to capture roles as detailed as possible, on the other hand generalization is necessary to establish groups of individuals providing exchangeable capacities.

When existing personnel relations in an organization are identified, it has to be determined whether they are suitable for mapping one-by-one to an organizational model or they turn out to be too inflexible, ambiguous or incomplete. For example a statement "most people who having spare time work on the integration project" is not helpful if its formalization yields the assignment of the whole development crew to that project. So

there has to be found a trade off between adapting the workflow management system's organizational model to the real organization and vice versa.

Once a complete organizational model is built, it is being used throughout the whole Emergent Workflow process: The originator of a new workflow fragment uses his organizational status to narrow the dictionary down to a subset which is relevant for him. A new process fragment can be assigned to its related process phase, team and project. Such knowledge facilitates also the composition of fragments and their placement in the current process. Just like any conventional workflow management system, an organizational model determines during run time which user is suitable to do a task and places it into his worklist handler. Finally, the search for templates in the repository is strongly supported by an organizational model analogous to the search for keywords in the dictionary.

2.3.5. Time management

Process definitions express control or data flow between activities and objects. They yield relative temporal dependencies such as "activity A can run concurrently to activity B" or "document D has to be processed before report R can be created". However, they do not tell anything about quantitative temporal constraints which are involved in any kind of process.

Example 4. Quality gates (see Figure 1.4 on page 8) in the automotive development process are an example for quantitative temporal constraints. They tell that a certain stage of features and quality has to be met until a certain deadline. All activities preceding that quality gate have to be completed until that deadline.

In general a maximum or minimum duration for a set of activities or the earliest and latest start and end date for activities are common temporal dependencies within planning a process. Furthermore, during run time actual values for start, stop and duration are being filled in. This is necessary for the integration with external applications managing temporal constraints, such as collective calendar systems or planning software. As soon as activities have been passed during enactment, temporal alignment between real-life activities and their planning counterpart can be checked and stored.

Example 5. Table 2.1 gives an overview of fictious temporal constraints of an activity. All types of constraints (start, stop, duration) can be defined either relative to other constraints or absolute in time. Each constraint has two planning values (earliest/latest respectively max/min) and one value recording the real values after execution. Notice that planned constraints are not mandatory and the information they provide can be incomplete, redundant or ambiguous. The earliest start time and the lastest stop time do not have to describe the same value as the planned maximum duration. Consistency between them can only be expected from recorded real values after execution.

Constraint	Absolute dependency	Relative dependency
Start		
earliest	2004-01-13 12pm	after termination of activity A
latest	2004-01-20 12pm	1 day before quality gate Q
real	2004-01-14 1:32pm	after termination of activity A
Stop		
earliest	–	after start of activity C
latest	2004-01-21 12pm	before quality gate Q
real	–	–
Duration		
minimum	1 day	–
maximum	7 days	1 day longer than activity A
real	–	–

Table 2.1.: Example for temporal dependencies of an activity

These demands motivate the integration of a central *time management component* in the Emergent Workflow approach. It handles all temporal aspects of process models during modeling, execution and evaluation.

Prerequisite for centralized time management is the availability of timing information. This can be assured if temporal constraints become an integral part of the process metamodel. Before or during execution, earliest/latest respectively minimum/maximum timing dependencies are created which need to be checked during execution. These values have to be integrated with process models as well as with instances. While and after execution, real execution values are derived either from the runtime engine itself or from interaction protocols. Hence running and archived process instances have to integrated execution timing for instances and activities.

Externalizing time management has besides its benefits strong requirements concerning synchronization and integration. After the initial transmission of timing constraints of a process model or an instance, constant synchronization is necessary to keep time management, process creating engine and runtime engine updated. While time management propagates notifications about the passage of defined time events, opposite components keep time management updated about status and schematic changes of running process instances. Notice that time management itself is not concerned with reactions initiated by temporal events. As a consequence, the time management component can not interact directly with users because reactions to regular or exceptional temporal events are instance-specific.

Example 6. What happens if the quality gate has been reached, but one preceding activity has not terminated yet? Let us assume that at the process definition level a rule has been set up that, in case an activity missed a quality gate, the head of the responsible department should be notified. The time management component though can not notify the head of department directly, as it has to be decided on the instance level who the responsible department actually is. So time management notifies the *runtime engine* about the exceptional event in an activity. The runtime engine has information about the responsible user, finds his department and emails the head of department.

2.3.6. Process creation engine

Conventional workflow management systems come with a software tool which is used to design a workflow explicitly. Before designing and enacting an instance, dedicated process designers either textually or graphically create a model in this software tool and transfer it to the workflow engine.

This procedure is not entirely suitable for Emergent Workflow as it does not separate modeling and enactment time of process definitions clearly from each other. According to the use case in Figure 2.1, dedicated process designer do still exist: They produce process models which either initiate an emerging process or provide a coarse framework for the overall process. Instances of these process models are then altered or completed ad-hoc during run time. To support this step, Emergent Workflow has to provide functionalities to document user interaction implicitly.

The idea of a *process creation engine* is to incrementally derive a process definition including instance-specific data from user interaction[6]. These process definitions are formalized according to a chosen metamodel (see Chapter 2.4). The input is a collection of interactions of the complete workflow management system using its interfaces. Input data is commonly styled in a textual and sequential manner. It is referred to as an *audit trail* and – originating from multiple interfaces – composed by the runtime engine. The audit trail describes what all external instances that interact during run time intend to do or have done.

It can be further clarified what the outcomes of a process creation engine look like if one distinguishes when a certain piece of documentation was created: The objective for documenting an event depends on when it has been created relative to its execution: Any documentation can be either created *before*, in the *meanwhile* or *after* execution of the according activity. The moment of documentation does not only influence its purpose but determines also how the start of process creation is triggered. These relations are listed in Table 2.2.

If a record was created *prior* to execution, planning support as well as synchronization of future activities are interesting aspects for a user. Such would be the estimation of

[6] In literature, post-hoc process creation from log files is referred to as *process mining*.

Time	Purpose	Trigger	Example aspects
before	planning & synchronization	explicitly by user	start date input data
while & after	documentation & reuse	implicitly by runtime engine	output data stop date

Table 2.2.: Documentation purpose relative to its creation time

resource availability and the early detection of their shortage. In this case, the process creation engine is activated *explicitly* by user interaction. When a process definition is created for planning, that is, the preparation of future activities, there is no way for any part of a workflow management system to detect the correct startup time and corresponding planning audits automatically. If process models are created *while* or *after* the execution of according activities, they serve for purposes such as documentation and reuse. Unlike the former, the invocation of the process creation engine is here likely to be triggered *implicitly*. For documentation, any kind of activity records is immediately relevant as throughout execution of activities, information such as start/end date, involved data and resources is completed on-the-fly.

Example 7. Suppose this example for a planned activity: An interdisciplinary meeting is scheduled for 16pm in a conference room. The according memo is created in the morning and the Emergent Workflow system has been set up to inform all project members of the upcoming meeting. The system might also put a watch on requirements documents and notify project leader about eventual changes taking place before the meeting.

Both on-the-fly and after the event documentation rather serve as a documentary basis for later reuse or analysis. If documentation is created during execution of an activity on-the-fly, especially temporal aspects of activities might be of interest.

Example 8. The activity start time of the interdisciplinary meeting was already fixed prior in the morning, but meetings in this fictitious organization are always open-ended. So the information about the meeting's ending time must to be added after its termination.

These varying usage purposes create different requirements regarding when to run the process creation engine on which data. In order to support planning, a process creation engine needs to evaluate data which is indicating upcoming activities, such as outputs from schedules or project planning tools. That forms a coarse framework of work structure but contains usually no details beyond the planned activity, starting time and duration. As that picture changes throughout execution of activities, the process creation engine has to add sequentially more details to the pre-modeled workflow.

If documentation or later reuse aspects are focused, then the creation of process models is delayed until all addressed activities have terminated and complete information is

available. That raises the question which data is relevant and if it is possible to manage the invocation of the process creation engine automatically. The recognition of relevant data for a certain usage purpose needs supportive data. That comprises state information of an activity as well as contextual data. Both tells whether due to the termination status detailed information is available and what the overall task – according to context such as a product number – of the particular activity was. In order to determine the right time to start a process definition extraction, a continuously running process creation engine is required. Otherwise an explicit start/stop mechanism of the process creation engine would be needed, which would turn the process creation engine basically into a cross-application macro recorder. Such solution would be impractical as it reduces usability drastically and it results effectively in explicit documentation of tasks. The purpose of a process creation engine is to avoid exactly that requirement.

As already mentioned, the sole recording of events caused by activities is not a sufficient input for the process creation engine to function properly. On the one hand, even repetitive tasks have differences and cause instances of the same process model to differ from each other. On the other hand, in real life unforseeable things can happen such that the planned course of activities gets interrupted or changed. The key for the development of an understanding for individual case variations is an extended view on an activity and the following related factors:

- Activity
- Classification
- Context
- Reason

Activity Documentation of an activity means to describe all relevant parameters which influence its execution during run time and all parameters which are affected by the execution. They can be identified as the following ones:

- Author
- Data input/output/modification
- Activity status
- Activity start/end time/duration

In order to find dependencies within an audit trail, first of all any record requires a note who its *author* is. This information is needed to determine whether it was an individual who created the entry or a whole group of either cooperating or independently acting

users. Based on that assignment the engine can estimate how many instance fragments can be extracted and what piece of information fits into which fragment.

Most activities involve data processing, creation or consumption. These external contacts are a substantial part of a documentation, consequently any form of *data input, output* or *modification* is vital to build a formal data flow representation. That includes objects such as paper documents, electronic documents as well as data objects being exchanged between applications, database queries or transactions within an ERP system.

Example 9. If in our automotive development example an activity "interdisciplinary meeting" in project "integration" (see Figure 2.12 on page 25) is scheduled by a project leader, the side door and chassis requirements documents will be used as a data input and the output might be new change requests.

As the audit trail is concurrently created with the execution of activities, the status of a running process instance has to be found out. That is based on the status of each single activity within the process; consequently the *activity status* is an integral part of an activity description. The status has to conform with the run time process status metamodel as shown in Figure 2.17 on page 61.

For planning and documentation purposes, temporal aspects are highly important as already motivated in Section 2.3.5. Thus, *start, end dates and activity duration* are recorded and are used within the process creation engine for process model creation and can be forwarded to the *time managment component* as needed.

Classification A classification of activity instances makes sense due to expected deviations of the "regular" case. Once recorded, the instances of an activity will look all alike if an annotation is missing on how the occurrence of an activity has to be judged. If an exception occurs only in 1 % of all activity records, a process model that weights an exceptional case equally likely to a regular case is misleading. A simple classification to avoid that is to distinguish between regular and exceptional activities.

Additionally, an exception which leaves out parts of the control or data flow is to be considered separately. Namely, if an exceptions behavior is to step over a commonly executed activity for some reason, then no trace in the audit trail would indicate its existence. To avoid that lack of information, for example an additional type of activity "replacement" might extend the exceptional classification. This relation can point to left out activities and indicates what the exception's character.

Context Contextual information describes basically any condition which is crucial for the execution of an activity. It can be either a side note or a further specification that subdivides an activity into distinctive cases. When a workflow user carries out tasks

as a particular role, this can describe a distinctive context as well as involved key data which determines the type of work.

Example 10. When a software tester performs the activity "basic module unit testing" in the context of a stage "first generation", then the activity has other characteristics than being executed in the context of "pre-release generation".

Reason "Why did we do it that way?" A workflow user might ask himself that question when he looks at past executions of activities. The parameters mentioned above already give a detailed testimony of what happened. Actually the reason is a formalized causal conclusion drawn from all other parameters. In order to make it easier to catch why something happened exactly the way it did, it should be mentioned explicitly in an audit trail. This part makes most sense in special exceptional cases, where strong deviations from more common procedures have occurred. In cases where the same path has been chosen as ten times before or the taken actions are made clear by contextual information and common sense, a reason is not mandatory.

Advanced but important issues are selective process creation and handling of erroneous and incomplete audit trails. As a workflow management system is in a central position handling many users and being integrated in big scale information systems, only a small subset of the information available at a time is relevant for creating a process fragment. Therefore the extraction procedure should offer parameters to control what record types are to be considered from a single logical workflow. It should also be able to compensate with non-conform inputs such as erroneous or incomplete audit trails. Especially having incomplete input is a very likely scenario if parts of a workflow are documented for planning early, but many details are missing and are supplemented piece by piece later on.

The **quality of process definition fragments** is another factor determining requirements of a process creation engine. All created fragments have to conform with a chosen meta model (see Chapter 2.4). This implies that there must exist specifications as well as methods to test the correctness of produced fragment. Dependencies within the control flow and the data flow have to be detected and modeled accordingly. To maintain robustness and a modular structure within the set of process fragments, there should not exist any implicit correlations or dependencies between fragments.

When a fragment has been created, it can be stored in the repository. Notice that it is vital to attach either manually or automatically a description of the procedure that is represented by the fragment. As one does not only want to save the fragment but also needs to find it later on within a potentially large set of fragments, a fragment's description is almost as important as its contents. A set of attributes like an identifier, a description, involved groups, associated process stages, number of activities and some examples for relevant information which is combined into a descriptive tag.

2.3.7. Runtime engine

The *runtime engine* is the central functional component of a workflow management system. Its invocation starts with the instantiation of process definitions. During execution of instances, they traverse state changes which trigger activities integrated in data and control flow. These activities are distributed and performed by external entities. The following aspects of a runtime engine will be considered in this Section:

- Interfaces
- Audit trail
- Task assignment with role management
- Instantiation
- Flexibility
  - Change classification
  - Flexible execution
- Annotations
- Consistency and correctness

Interfaces Looking at the inputs and output objects of a runtime engine in a black box manner, one notices that interfaces 2 through 5 introduced by the workflow reference model (Figure 2.7 on page 16) refer to the runtime engine: Client applications receive work items (interface 2), applications are invoked during run time (interface 3), other workflow enactment services exchange objects during during process execution (interface 4) and administrators monitor the progress of instances (interface 5).

Input objects for a runtime engine comprise process models from the repository, the process creation engine or an external process modeling tool and interaction protocols of its interfaces to client applications, server applications and external process enactment services.

During run time, a runtime engine outputs task assignments to client applications, monitoring information to administrators, and an audit trail for storage and reuse purposes (e.g. to the process creation engine). Furthermore, it exchanges status updates and synchronization messages with external workflow enactment services.

Audit trail The term *audit trail* refers to a continuous stream of use data in a machine processable form. This stream can either originate from a workflow management system (inside-out) or is handed to it from external information sources (outside-in).

Conventional uses of an "inside-out" audit stream are monitoring and controlling functions for workflow participants in order to extend their own scope on the processes they work on. Monitoring and controlling may be used also for compensation of the lack of awareness inherent in workflow implementations. An audit stream going outside-in helps administration and management to monitor and control procedures and gives them a better understanding of operational dynamics. But also trading partners or customers can use monitoring functionality for optimizing B2B collaboration (e.g. supply-chain forecasting) or tracking of remote processes (e.g. order tracking).

A special use of Emergent Workflow for an audit trail is as an input for the process creation engine. However, "raw" data entering the interfaces of Emergent Workflow is not yet suitable for it. First, as mentioned in Section 2.3.6, process creation imposes strong formal requirements on its input as well as filtering abilities to receive an audit trail selectively. Numerous input sources deliver massive amounts of protocol data consisting of events, data actions, transaction information and others. All of them arrive in different data formats. As all that information arrives at the runtime engine, its responsibility is to filter incoming data, arrange it in a common format and deliver a selected stream to the process creation engine. The outputs of the process creation engine in exchange are planned for later reuse and are input for the runtime engine at a later point of time. This process is visualized in Figure 2.13.

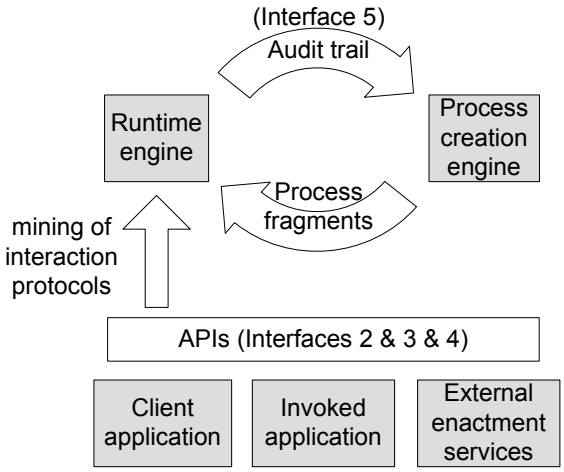


Figure 2.13.: Audit trail flow

External applications at the client and server side as well as external workflow enactment services communicate through different a layer of various application programming interfaces (APIs) with the runtime engine. With respect to the workflow reference model (see Figure 2.7 on page 16) that layer represents the interfaces 2, 3 and 4. Besides control data, use data is exchanged by that interface. Hereof protocols of workflow user interaction are extracted, which is also called *mining*. After being forwarded to the runtime engine, this information can be processed and spread to other components such

as the process creation engine. Although outputs of the process creation engine are not directly passed back to the runtime engine, in the course of process model reuse they return to the runtime engine. Thus, the stream of process fragments from the process creation engine back to the runtime engine can be interpreted as a return value to the audit trail, closing a cycle between these components.

An interesting, requirements-related architectural question is to consider whether a *push* or *pull* mechanism is realized between the runtime engine and the process creation engine. These patterns refer to how communication is initiated between data source and destination. The answer to that question influences where program logic for the assembly of relevant audit trails and the initiation of process fragment creation is settled. In this case a *pushing* architecture means to have the runtime engine to decide on timing and content of audit data sent to the process creation engine. That case implies a continuously running process creation engine, which service-like awaits incoming audit trails and answers these requests with the delivery of process fragments. In a *pull* architecture, the process creation engine requests an audit trail from the runtime engine by specifying when and what type of audit data will be transmitted. Obviously, here the process creation engine needs an external trigger which initiates the explicit query.

If one compares these two possible realizations with the listing of documentation time and purpose in Table 2.2 on page 29, one can detect a relation between the usefulness of either the push or pull principle and the purpose of documentation. If documentation is meant to be created *before* the execution of activities for planning & synchronization reasons, fragment creation by the process creation engine is explicitly triggered by workflow users. Hence, a *pull* mechanism would make sense where the process creation engine – comparable to SQL statements in relational database systems – requests excerpts of the overall incoming audit trail. In the contrary case of documentation during or after the execution of activities, an implicit run by the runtime engine suggests a *push* architecture.

Task assignment with role management Task assignment addresses an event during execution of a process instance and is in conjunction with the activation of activity instances. Activation is an intermediate activity state between being *inactive* and *running* (activity states are introduced in Section 2.4.2). Task assignment describes the resolution of an abstract role model into existing real individuals. Only interactive activities (e.g. activities with associated roles incorporated by either human users or software agents) are affected of this action as they have a role association. Automatic activities can be started immediately upon completion of all pre-constraints, thus they do not distinguish between the states *activated* and *started*. When an interactive activity instance switches from *not_activated* to *activated* and further to *started*, task assignment is done by the runtime engine. In order to find all personifications of a role within an organization, the runtime engine relies to an organizational model as described in Section 2.3.4.

Major objectives of task assignment are *optimal work efficiency* and *flexible assignment*

of work load. Optimum efficiency denotes a maximum average throughput of work items using the available resources while minimizing erroneous processing and administrative overhead. Throughput can be influenced by flexibility of assignment, such as automatic rescheduling of tasks from busy participants to idle participants. The automation of assignments supposedly reduces overhead but reduces also flexibility if realized without possibility of manual interference.

Two queuing models of task assignment are possible: *One virtual global queue* describes virtually one worklist shared by all workflow users embodying a particular role. What happens is that an activated activity shows up as a work item within the worklist handler of each workflow user with a corresponding associated role. As soon as a user chooses to process a work item, the activity's state changes to *started*. At this point, an instance of that role has been assigned to the activity instance. Concurrently, the work item is removed from the virtual queue and disappears from all other role personifications' worklist handlers. This type can either offer a list to each user from which he can choose a work item or all users can just request an anonymous "next" work item. Individually selectable items offer more flexibility but bear also the possibility of non-uniform item prioritization. By implementing a virtual global queue, the workflow management system can influence prioritization of activities by introducing priority levels into the queue. If tasks are assigned to users by an anonymous "get next" retrieval, this ordering is fixed. If users can actually see the contents of the queue and choose work items within constraints such as a minimum/maximum idle period of items, they can influence prioritization of work items.

With *multiple queues*, one queue is maintained individually for each workflow user. Upon activation time of an activity, it may be assigned to a specific workflow user and appears as a new work item in his worklist handler. This queuing method shifts responsibility for equal work distribution to the workflow management system. It offers a higher level of automation and reduces potentially more overhead. Furthermore it gives a workflow user a clear idea of the anticipated work load and facilitates individual planning. On the other hand individual queues decrease the level of flexibility. If a task has been assigned mistakenly or needs manual changes, an additional function for re-queuing work items is indispensable.

Independent from a queuing method, the moment of role resolution is flexible. At the earliest, it can be done during instantiation of a process definition, at the latest it has to be completed when the activity instance switches into the *activated* state. The earlier resolution is done, the better activities and future engagements can be planned. If process instances are rather long-running, then occurring changes generate very likely the need for updating resolution. These changes can originate from both sides, the organizational model and the process instance: Available employees switching positions or becoming unavailable as well as altered or stepped over activities are examples for such scenarios. In any of these cases the validity of existing assignments must be checked and is an elaborative task. The later task assignments are completed, the more likely they are stable until processing. Participants' flexibility though is reduced by late resolution as

incoming work items are "popping up" right away and can not be anticipated throughout a longer time frame.

Instantiation Sources that initiate instantiation are all input and output interfaces as defined in Section 2.3.7 on page 33. External events trigger the instantiation of process definitions through several defined interfaces: Using interface 2, the initiator is a workflow user who is sufficiently authorized to instantiate a particular type of process definition. Invoked applications are another source of process enactment using interface 3. They may be external software such as an Enterprise Resource Planning system upon the start of a new procurement transaction. Interface 4 integrates workflow management systems. This also includes that external workflow management system can not only exchange data or synchronize with their internal activities, but can also initiate the creation of process instances. Finally, also the administrator of a workflow management system can monitor and influence all aspects of instances including their creation using interface 5.

Example 11. A mechanical engineer receives a new change request from a colleague as changes collide that were concurrently made on the digital mock-up. As change requests are frequent events during the development process, a generic pre-modeled process definition exists. In order to start processing a change request, the engineer instantiates a change request process definition and executes the instance.

The first functional requirement is to check whether the instantiation of a process model is executable: Basically, an instance is executable if a start state and a terminal state are defined and they are "connected" by a sequence of valid state transitions. Furthermore, no invalid activities, roles or resources are allowed to be referred to by an instance. Second, the runtime engine usually applies an initial state transition on instances after their activation. That is, the execution is initiated by starting the first activity according to the instance control flow.

The fact that the runtime engine runs potentially many instances concurrently imposes nonfunctional requirements on it. Van der Aalst and van Hee identify a number of workflow bottlenecks [AH02]: First, the overall number of instances in progress can grow large. If there are many instances in progress, it may indicate an existing problem. Causes include major fluctuations in the supply of instances or resources being to inflexible or weak dimensioned for heavier use. However, it may also be that the process contains too many steps that need to be passed through sequentially. Furthermore, completion time of instances could be too long compared to actual processing time. The actual processing time of an instance sometimes forms only a small fraction of the total time when it is in progress. If this is the case, there may be a whole range of possibilities for reducing completion time. Moreover, the level of service can be too low. A workflow's level of service is the degree to which an organization is able to complete instances within a certain dead line. If completion time fluctuates widely, then the

organization offers a low level of service. In that case it is not possible to guarantee a particular completion time. A low level of service also exists when there are many "no sales" occurring – potential instances can not run because waiting for progression within the runtime engine will take too long. When a user knows that it will take a long time to complete an instance, he will try to circumvent the process. A low level of service can indicate a lack of flexibility, a poorly designed process or a structural lack of capacity.

The symptoms mentioned above point to possible bottlenecks. To identify them one needs to benchmark values for these measures, for instance from comparable processes. Usually, it is not sensible to combat the symptoms using only emergency measures but to tackle their causes.

Flexibility Static workflows are easy to handle, but fail in scenarios as motivated in Section 1.1. As flexibility is an issue of particular interest in the light of Emergent Workflow, this paragraph is actually subdivided into several points of view: First, the flexibility is the requirement emerging out of the need for *change*. Hence, the first subparagraph will introduces ways to characterize changes on different workflow perspectives (see Section 2.4). Next, the term of flexibility will be broken down into more concrete measures that allow variable kinds of deviations.

Change classification Changes during run time arise because parts of the information that constitutes the workflow are not known during build time or changes occur while the system is in production. Van der Aalst and Jablonski propose the following classification [AJ00].

In order to classify, what types of focus exist when managing changes, a number of change dimensions are introduced:

- **Maintenance of correctness and consistency.** This points at potential errors resulting from change, which can be either syntactic and semantic errors. A semantically correct process instance is able to reach a terminal state without any errors or deadlocks.
- **Single-perspective and multi-perspective errors.** With respect to the workflow perspectives, errors are identified that affect either only one workflow perspective or multiple perspectives at once. A deadlock is only visible in the process perspective, whereas a task pointing to nonexistent roles and data objects occurs in the organizational and information perspective.
- **Transient and permanent errors.** Errors caused by changes can last for different amounts of time. Transient changes exist only temporarily and do not affect new instances. Permanent errors are lasting longer and affect newly created instances as well.

When solutions are proposed to implement changes and resolve errors, one can distinguish between introducing *flexibility by configuration* and *flexibility by adaption*. The former offers more powerful design constructs and integrates changes into the meta-model. Flexibility by adaption tries to limit changes, manage multiple versions and avoid errors by the application of inheritance concepts.

Introducing flexibility means to allow certain types of changes. These types can be classified by the following six characterizations:

1. **What is the reason for change?** Reasons may be located in the context of process execution outside the system like changing requirements or technology, but can be triggered also from the inside of the system such as errors and problems causing failure.

2. **What is the effect of change?** On the one hand, *momentary changes* influence a limited set of instances. They occur typically as the result of errors or exceptions and pass by without permanently altering the process definition. On the other hand, *evolutionary changes* take action for all instances starting at a certain point in time. Their type of change is rather structural and more permanent such as a changing legislation that eventually changes the process context.

3. **Which perspectives are affected?** The type of change is reflected very well by the related workflow perspective (Figure 2.14 on page 54). In addition, deletion or modification of process definitions including their tasks and routing are typical changes appearing in the *process perspective*. Staff changes and other modifications of the organizational structure relate to the *organizational perspective*. In case data structures are added, removed or modified, these changes become evident in the *information perspective*. The *operational perspective* shows the exchange of invoked applications and other operations related resources. If finally linking points between the perspectives such as task assignment are subject to change, these and only these changes will be reflected in the *integration perspective*.

4. **What kind of change?** This refers to the way a change operation affects the functionality of a process. As control flow oriented changes deal with the alteration of tasks and their structural arrangement, functionality can be *extended*, *reduced* or *replaced* by adding, removing or replacing a task. If the dependencies are just rearranged between existing tasks, the change is called a *re-linking change*.

5. **When are changes allowed?** A change is either allowed *at entry time* only or *at any time*. The entry time denotes the very moment an instance's specification is set up for each involved perspective; after that moment all specifics are not allowed to change any more. Otherwise, changes are allowed at any point during workflow execution on-the-fly.

6. **How are existing instances handled?** A number of alternatives exist for how running instances may be handled after a change operation. A *forward recovery*

aborts old instances and compensates them outside the workflow management system. *Backward recovery* aborts, compensates or rolls old instances back in order to get them restarted with new definitions. Alternatively, one lets old instances *proceed* as they continue running the old way. Only new cases are instantiated with respect to the change. A *transfer* operation migrates old instances to new process definition, whereas a momentary *detour* allows the change to settle before actions are taken.

Three frequently named change types *exceptions*, *ad-hoc workflows* and *dynamic changes/ migration*, will now be categorized using the first five criteria given above:

Exceptions are usually unexpected events which are caused by failure of some component rather than deliberate changes. Reasons for exceptions are mostly located inside the system, they have momentary effect on a limited number of instances and affect the information and operational perspectives. Functionality is either reduced or replaced by exceptions and they occur at any time on-the-fly.

Ad-hoc workflows are edited shortly before and during enactment on an instance level. The reason for ad-hoc changes is located outside the system and changes have only momentary effects. Although any perspective can be affected by ad-hoc changes, mostly the process perspective is focused. Ad-hoc changes can extend, reduce, replace or re-link functionality of a process instance at any given time during execution.

Dynamic changes/migration deals with handling of instances running on an old process definition after the process schema has been changed. This is not always straightforward, e.g. the new model may not have an execution state corresponding to the state of the old instance which was specified by variables indicating which tasks have already been executed. Reasons for migration are usually irrelevant and by modifying the process definition, they apply evolutionary changes. They have an impact on all perspectives and perform any kind of change as well. Only on-the-fly changes have to be investigated as entry time changes are considered straightforward: It can be assumed that any new process model has a correct initial marking state.

Types of flexible execution As already mentioned, flexibility during execution can be created by applying various measures. In the following enumeration, types of flexibility are classified according to their degree of flexibility in time and are further elaborated in the following paragraphs.

- Schema evolution
- Late modeling/Case handling
- Ad-hoc changes
- Exception

Schema evolution describes schematic changeability by iterating a design phase, *late modeling* predefines limited short-term flexibility on details of process definitions. *Ad-hoc changeability* constitutes spontaneous changeability of the execution state of process instances. *Exception and case handling* provides means for spontaneous change of state of process instances.

Schema evolution or evolutionary modeling refers to incremental changes applied to process definitions (compare Wargitsch et al. [WWT98]). Instances of explicitly modeled process definitions are observed by process designers and improvements according to analysis outcomes are integrated into process definitions. This method contrasts *process reengineering* where the entire process is radically redesigned to achieve performance improvements (compare Davenport and Short [DS90]). This procedure adapts to the workflow life cycle as depicted in Figure 1.1 on page 2. Thus flexibility is provided for long-term changes, however it is not helpful for short-term flexibility as mentioned in Section 1.1. In order to enable process model evolution, process designers require methods that allow them to apply schematic changes to a model such as the insertion/removal of a activity or the alteration of the control flow. Subsequently, running instances have to be handled in one of the ways mentioned in the previous paragraph on change classification. Most desirable is the solution to migrate instances to the new model by either changing their schema on the fly or restarting them and auto-execute them until a state that was defined equivalent to the originating state.

Late modeling/Case handling addresses incomplete modeling with *unstructured process portions* which are also called black boxes or placeholders (compare Herrmann et al. [HSW97]). *Late modeling* means the replacement of placeholders with spontaneously modeled sub-processes during run time. This information gap has to be filled up during run time in order to let the process instance terminate correctly. If a workflow user can choose at run time from a number of previously defined alternative process fragments referred to as *cases* in order to replace the black box, a *case handling method* is applied (compare Hagemeyer et al. [HHJHS97]).

If process definitions containing black boxes should be executable, unstructured process parts have to be identified and marked adequately during process design phase. Eventually they are equipped at design time with a *case base* which describes several alternatives for structuring the black box upon activation. The runtime engine needs to make sure that each unstructured process portion is submodeled before it can transfer the activity in the state *started*. As a subgraph is modeled individually for each instance by the workflow user in charge, the user also has to examine the case base for a suitable case that matches the individual context. If such does not exist, then the ability to alter existing cases and to add new cases to the case base is required. Not all activities are meant to be arbitrarily changeable by workflow users, consequently a classification of flexibility for activities has to be established in the process metamodel (see Figure 2.16 on page 58) and implemented during process design. If a workflow user decides to modify an existing case or to introduce a new case, this action influences secondary related activities. Such would be dependencies like a removed data output which is expected by

another activity. Coordination and propagation of subsequent changes is a task which needs functional support by the runtime engine. For each activity, it must maintain a list of dependent activities and their processing role instances.

Late modeling offers the benefit of short-term flexibility without reiterating through process design. However, spontaneous changes are restrained to process parts where short-term actions were anticipated and unstructured process portions with case base were either realized during process design phase or are spontaneously created during run time.

Limitations apply when process enactment deviates from planned flexibility because an unexpected situation has occurred. Late modeling does not offer sufficient functionality to formalize handling of exceptional situations.

Ad-hoc state changes are meant to apply instant changes to default state transitions of instances: An activity can be skipped, moved, inserted or removed. Execution can return to the previous activity, reset or step over the current activity. These modifications do not influence the process model but are restricted to a specific instance.

Each ad-hoc state change potentially endangers correctness as a change could make a terminal execution state unreachable. The responsibility for avoidance of such "bad" changes is carried either by the workflow user applying the change or by the workflow management system. The latter case requires nontrivial process analysis which validates the change: As activities are correlated (by control/data flow, usage of resources, ...), manual changes may interfere with pre- and postconditions of activities. They might require successive adaptions of other activity states to prevent unwanted states such as deadlocks. Hence, checking and modifying mechanisms for process instance states are required.

Exception handling A computer-based workflow management system has its strengths in structuring, rationalizing and routinizing work. The fewer unscheduled manual intervention is required, the better is the system's performance. *Exceptions* are defined by Strong and Miller [SM95] as follows:

> *We define **exceptions** in computer-based information processes as cases that computer systems cannot process correctly without manual intervention [which is] a definition broader than "errors".*

One can distinguish three major perspectives on exceptions:

The *random-event perspective* on exceptions addresses situations which occur infrequently, are non repetitive and have random character. While it is assumed that a workflow management system works correctly most of the time, little can be forecast about exceptions. Such might be caused by external influences like power downtime or physical damage that harm information systems as well as internal malfunctions. Due to their unpredictable nature there is no efficient way of resolution for these kind of ex-

Perspective	Underlying assumption	Solution approach
Random-event	Exceptions are unpredictable	None
Error	Errors (from operations, design, changing environment)	Eliminate causes
Political system	Political system causing conflicting interests	Efficiently detect and handle exceptions

Table 2.3.: Perspectives on exceptions (compare [SM95] Figure 1)

ceptions. Depending on the negative impact of specific types of exceptions, precautions may be taken in order to minimize their probability.

The *error perspective* looks at exceptional situations caused by errors in operations, process design or changing environment. *Operational* errors are most common when human interference with input or output is handled incorrectly or the user misunderstands the interface or system. Erroneous behavior can also be traced back to *weaknesses in system design*. The process models can reflect the real process incompletely or incorrectly. That type of error is likely to exist due to many factors influencing correct functionality, fuzzy knowledge about true processes and the problem's high overall complexity. Additionally, errors are introduced by *changing external requirements* caused by a flexible environment. As an information system does not evolve as smoothly as real processes which it depicts and supports, over time the electronic process diverges from the real process. Differences cause increasing errors, because the workflow management system ends up processing a process it was not designed for. While operational and design errors are conceptually tough to avoid, frequent minor adaptions and evolutionary changes to process models reduce errors caused by a flexible environment. In this context, the term Total Quality Management[7] (TQM) is often mentioned. It describes a management methodology trying to detect the causes for primary error sources and to eliminate them.

One can conclude from the estimations given above that exceptions are a regular part of process flexibility and require to a certain extent efficient detection and handling support. The error perspective mentioned last is the most likely error type to be encountered in Emergent Workflow. As high flexibility in the addressed field of application is likely, the occurrence of an exception in this context does not mean that such an event is exceptionally rare, but that exceptions occur with many variations – they are legitimate special cases.

Formally, exceptions are arbitrary ad-hoc deviations to any workflow component at run time. Any workflow perspective (see Figure 2.14 on page 54) can be affected by exceptions: On the process instance, instantiation, execution or termination of process instances can be interrupted by exceptions. A changing organizational model causes potentially exceptions as well as problems with data objects being manipulated dur-

[7] Compare http://en.wikipedia.org/wiki/Total_quality_management

ing execution. The same applies to client and server application integration or other resources.

With respect to the definition of exception given above, *exception handling* denotes manual interventions in Emergent Workflow procedures which resolve or compensate an exception's effects. In fact, exception handling splits up into two distinct activities: *Detection & information* and *handling*.

First of all, it is necessary to create an awareness within the workflow management system for an exception and to propagate that information. Therefore one needs to *detect* an exception and its type. Exceptions can be caused by external events which are not system-related or of technical nature. These kinds of exceptions have to be entered by an external entity such as a workflow user or a software agent. If for example the user interface of the runtime engine offers an explicit entry form for the description of exceptions, a reaction can be directly declared by the user as an exceptional state transition. If not notified from the outside, the runtime engine has to recognize from unexpected situations or other indicators that an exception has occurred.

If the exception is system-related and caused by an event within the workflow management system, then an exception message has to be broadcasted in order to notify other components. An example would be the alteration of the organizational model during run time. As the organizational model changes, the re-assignment of tasks for running instances becomes necessary, so the runtime engine should receive a message about this. In return, the runtime engine can come up with a delegation rule and reschedule waiting jobs in other worklists if possible.

Example 12. In order to compensate an exceptions caused by a failed activity, a workflow user can handle the exception by a manual intervention in one of the following ways:

- **Ignore the exception.** This is the most simple way of exception handling which might be helpful under certain conditions.
- Retry the failed activity. This makes sense if failure was caused by a momentary reason which has changed.
- **Perform a partial rollback.** With this option, one can try to circumvent the execution path that lead to the exception. A partial rollback means to undo or compensate a number of previous activities until a branching state is reached. From there, an alternative path can be chosen that leads to a terminal state without touching the failed activity.
- **Add extra activities for compensation.** Execution continues after the failure, but an extra activity is inserted in the future process that compensates previous failure.
- **Delete planned activities.** If there are succeeding activities that rely on the failed activity (e.g. they need its data output), then the solution could be to delete all dependent future activities and to proceed with execution.

Annotations Annotations are supplemental records created during run time by workflow users. The idea is to give workflow users a tool to annotate the execution history of a particular process instance. The addition of an annotation does not interfere with the schema of process definitions, but is a user-based tool to distinguish a certain case within a case type. It evolves from the user's perception of an individual contextual situation. If certain conclusions can be drawn from the context and are valuable for later reuse, the user may quote it accordingly.

Example 13. Suppose during implementation of a software component, a software engineer realizes that an issue should have been tackled during component design and is causing unnecessary work right now. In the last development cycle, the same problem had shown up, too. So it would be nice to give the engineer a tool to formalize his idea because otherwise it may be forgotten until the next cycle. Of course, he can put down a note in his notebook or email his project manager about it, but this will not make his idea lasting and broadly available. If in fact until the next development cycle team composition changes, his idea may get lost. So it would be helpful if Emergent Workflow would offer means to annotate instances or activities – in this case the component design activity – which have already been terminated during run time. By adding such notes or modifying existing information, reference knowledge about an activity is increased and it can be used more intelligently when the model is re-instantiated or used as a template for another model.

Correctness In order to avoid errors during enactment, the runtime engine should take as much of the responsibility of assuring correctness for process models and instances as possible. In the following paragraph, *semantic* and *syntactic* correctness are distinguished.

Syntactic correctness of process models is available if *consistency* and *completeness* of process models can be assured[8] If all elements within the model notation are sufficiently described in the metamodel, a model is consistent. Completeness can be guaranteed if all mandatory constructs from the metamodel are integrated in a process model.

Data in a workflow management system is called consistent, if all integrity requirements are met. Each process instance is supposed to correspond to one associated process definition. Upon changes to the definition, the conformance of all associated instances has to be assured in order to maintain structural identity with its associated definition, e.g. by migration. If only a subset of all running instances of a process definition is intended to be adapted during run time, then the remaining old instances may be associated to exclusive old copies of the process definition.

For a runtime engine that allows multiple flexible operations such as late modeling, schema changes and ad-hoc modification it is nontrivial to uphold consistency and correctness. This will be subject for discussion throughout the rest of this thesis.

Semantic correctness addresses whether a built model is able to function semantically as intended. Typical examples for semantically incorrect models are models whose instances can not be executed or do not terminate correctly upon execution. *Reachability of termination* issues that an process instance is only executable if its start and termination state are defined and are "connected" by a sequence of valid state transitions. Correctness is here given if *any* sequence of transitions beginning from the start state leads into a valid termination state.

The correctness requirement for process definitions addresses their behavior after changes during run time. As structure and dependencies are getting changed and instances are being migrated, the runtime engine has to run checks on them to make sure they are still able to reach the designated termination state. Alternatively, changes are only allowed in a way that – in conjunction with an appropriate metamodel – does not harm correctness such as in ADEPT [Rei00, RD98]. Possible incorrect behaviors can be a number of state transitions which lead into a deadlock or an infinite loop. These states do not contain a correct termination state. Other problems after alteration of process definitions can be a lacking reachability for activities or unforeseen termination.

[8] Compare zur Mühlen [Müh96] p.17 et sqq.

2.3.8. Process matching engine

Central ideas of all data processing in Emergent Workflow are documentation and reuse of previously defined structures (compare Section 1.2). The *process matching engine* is a necessary tool to accomplish the idea of reuse.

All interfaces of Emergent Workflow and components like the process creation engine are busy with internalizing external data. As a consequence, a massive collection of audit trails, fragments of process definitions, their instances and compositions is accumulated. Emergent Workflow is likely to be used in an environment that requires adaptability to changing conditions. That implies that the amount of slightly differing fragments grows rapidly.

The process matching engine is supposed to support different user groups in finding information from the repository. A workflow participant wants to find process fragments from previous instantiations in order to build the current instance execution on a template. Administrators who monitor actions on the runtime engine want access to the latest pieces of the audit trail. Process designers want to obtain stored process types, archived instances, compositions and audit trails for analysis and improvement.

Input & output characterizations On the system side, the process matching engine accesses the repository which holds all available data structures (see Section 2.3.9). These are stored in databases for each type and equipped eventually with helpful access constructs such as an index. Complex data structures such as graphs are supposed to have attached tags containing important search criteria. The process search engine must be able to read all data structures in the repository. Any request is answered with a (possibly empty) set of return elements matching the search.

Theoretically, database systems used for the repository already provide access methods for their contents which could be sufficient for Emergent Workflow, too. The reason why a designated search engine is proposed lies in the fact that search types required by Emergent Workflow exceed common database search methods' abilities. All types of users or their client applications interact with the process matching engine by submitting requests that characterize repository elements. These requests contain a number of constraints as well as supplementary data objects to characterize their expected result set. Query constructs offered by common database systems are not able to cope with similarity matching of data object.

Let us focus in the following considerations on the search for fragments of past executions.

Example 14. A query for a process fragment using constraints and expressed in natural language could be "Show me all fragments that have been created by mechanical engineerings using the activities A, B and C since two weeks ago and sort them by descending date".

In this example, the constraints refer to a number activities and to information that was collected during enactment. Typical questions for instance-specific characteristics would be:

- What activities were executed?
- Which data streams and functions are included?
- What organizational entities and applications are involved?
- What was the duration of each executed activity?
- Which disciplines were involved?
- What was the process frequency in the past?

A query can be supplemented by data objects which describe what a return object should look like.

Example 15. A process designer has a process fragment and wants to find out if this process fragment occurs frequently. He passes it to the process matching engine along with a query "Show me all fragments recorded during the last two weeks which are similar to my process fragment."

The first query presented in the Example 14 gave an exact type and number of constraints that all result have to comply with. In Example 15, a constraint is given along with an fuzzy description as a query. Similarity between fragments can refer to either *syntactic* (same activities, users, data), *semantic* (same function and effect) or structural (process graph structure) *similarity*. In this case a result list is expected where the most relevant (similar) match is presented first, followed by less similar matches in decreasing order. Obviously, different kinds of searches require different matching processes.

Matching process Without specifying a particular matching algorithm, different types of algorithms are required for matching according to different search types:

First, queries requiring exact matching are to be differentiated from those requiring approximate matching. *Exact matching* is characterized by a number of quantitative constraints which can be composed (e.g. with boolean operators) to a complex expression. Each repository element is checked for accordance with the expression and either matches it (and is put into the return set) or does not. The exact matching algorithm returns a finite set of matches on the query. *Approximate matching* is needed when the query contains qualitative constraints such as similarity aspects. When a qualitative constraint is used for searching, the result is never absolutely clear but represents a relative rating of matching quality. When performing a similarity search with a given

reference as parameter, the only absolutely "safe" matching is obtained when the found object matches exactly and equals the search parameter. Otherwise, a rating based on a similarity metric is added to each matching that indicates its quality. A user query based on qualitative constraints expects a return set of those matches that yield the highest rating. Notice that without any filtering, the result would be always the complete set of searched objects available, because any repository element receives a (possibly low) rating. Therefore, a *threshold* needs to be either determined by the process search engine or is specified by the user in order to cut off results whose matching quality is too low.

With respect to the application domain of searching process-related objects, a further distinction can be made between requests that require descriptive searching and those involving a schema-matching search. A *descriptive search* contains constraints that can be checked without an in-depth analysis of process structures. Rather, each process element inside the repository holds a descriptive tag which roughly classifies it. Such would be a creation time, the creating user and the overall context. Descriptive search is supposed to be rather simple and quick. *Schema-matching search* denotes searches asking for details, which are not contained in descriptive tags but have to be obtained using more elaborate structural analysis of repository objects. Typically, fuzzy queries causing approximate matching rely on schema-matching search.

Having mentioned more and less elaborate matching processes, it is worth to reinforce the observation that efficiency plays a major role for algorithms implemented in the process matching engine. Searching through a potentially large number of process objects and matching them with complex constraints including structural comparison is a demanding task for computer hardware and software. However in most situations when a process search is invoked, a user does not want to wait for results longer than a short amount of time. Consequently, a trade-off between functionality and performance has to be found for a useful implementation of process matching.

2.3.9. Repository

The repository has already been referenced frequently as all other components' activities are accessing it. After discussing all other components of Emergent Workflow, it becomes evident that basically all kinds of information are stored either temporarily or permanently. To make that happen, all components rely on a common *repository* for data storage and retrieval. In this section, no particular data structures are proposed due to the high-level characterization approach and the following characterization of the process metamodel in Section 2.4.

Storage Different kinds of data are stored either temporarily or permanently. **Temporarily** stored data is used to depict and update the current state of the workflow management system. As this topic becomes quickly implementation-specific, We will not go into much detail on this matter. It may be only said that the core of tempo-

rary data are *states* and their *transitions within the runtime engine*. It runs multiple instances, all of which have different types and states. If ad-hoc changeability is allowed on an instance level, supplementary data is attached to instances, indicating and defining the change operation. *Time management* is closely integrated into execution of instances as it sets and checks temporal dependencies. This kind of temporary information changes consistently, frequent updating read and write operations can be expected on it during run time of the workflow management system.

Permanently stored data serves the purpose of preserving and building a collection of useful knowledge for a longer period of time. In Emergent Workflow, this includes especially traces of current processes and any supportive information for reuse. General knowledge like the *dictionary* is stored permanently as it preserves a depiction of the commonly used vocabulary. Also the *organizational model* is a permanent system representation of an organization. It contains a hierarchy of roles, assignments into groups and associations of roles with real personnel. The fundamental part of reuse-oriented, permanently stored data are *process models*. Descriptions of process model are optionally supplemented with a classification of granularity that describes its level of detail. Furthermore, the allowance of schematic changes on process models extends their representation with versioning information, as the schema of a process type changes over time. Parts of temporarily stored data as described above becomes permanently stored data. *Instance fragments* created by the process creation engine are archived in the repository for reuse, such as the establishment of a case base (see Section 3.1). Also *compositions* of fragments created by process designers are stored permanently as they were created for the sole purpose to enable post-hoc analysis. Finally the source of process fragments, *audit trail* is also interesting for permanent storage to a certain extent. As audit data represents the most quickly and a permanently growing amount of information, practice has to show whether it is meaningful and possible to store the full amount of audit data permanently and efficiently.

As major amounts of data are collected and created inside Emergent Workflow, data structures for storage may be chosen with an eye on space efficiency. On the other hand, a convertible and open representation would be recommendable for better reusability. The Workflow Management Coalition proposes for example XPDL[9], an XML Process Definition Language which offers a metamodel and an exchangeable representation form for process definitions. For use with Emergent Workflow, this format may be used if constructs for flexibility requirements are added. Thereby, process models would become easily exchangeable but also space inefficient due to the high verbosity of XML which makes it a questionable choice for permanent storage. The same issue holds for possible representations for the audit trail such as the XML workflow log format proposed by van der Aalst et al. in [ADH+03] or an instance-level case representation proposed by Madhusudan et al. in [MZ03].

[9] See the XML Process Definition Language Specification Version 1.0 Final Draft: http://www.wfmc.org/standards/docs/TC-1025_10_xpdl_102502.pdf

Access methods are the necessary counterpart to data representation within a repository to enable reuse. They describe ways to receive read and write access to all use data. As any kind of repository is most likely based on database technology, basic querying mechanisms as well organization forms for structured storage are already available. That includes organization forms such as tree structures, hash tables or indexing and will not be elaborated here any further. Notice however that a textual representation of process structures is neither very "handy" nor very expressive in text-based data structures. Thus, it is suggested to supply process fragments and compositions with a *textual tag* containing a description that can be used for most common search criteria. Information such as the creator of a fragment, its start/stop date, its type and more can be easily derived from the context when archiving a terminated fragment. The same holds for compositions, here the tag could be composition of the tags of all contained fragments.

2.3.10. Requirements summary

In this Section, component-specific requirements are summarized in a tabular representation. For each component the source or kind of input and output are giving and indicated by an "I" and "O" in the left column. Then an enumeration of the most fundamental properties is given. Each property is associated with a unique identifier located in the left column, such as (UI2). These identifiers are used in Chapter 3 to refer to a property match between Emergent Workflow requirements and related work.

User Interface/Client Application/Agent	
I/O	Exchange of control and use data between runtime engine and a human-machine interface/an agent
(UI1)	– Functional specifics for user groups
(UI2)	– Usability (intuitivity, simplicity, documentation)
(UI3)	– Configuration & customization
(UI4)	– Creation of accurate/detailed interaction protocols
Server Interface	
I/O	Communication with external applications/workflow enactment services, runtime engine
(SI1)	– Standardized interfaces for synchronous/asynchronous communication with external applications and workflow enactment services
(SI2)	– Support different levels of interoperability
(SI3)	– Creation of accurate/detailed interaction protocols
Dictionary	
I/O	Dictionary contents are communicated with all other components
(D1)	– Establishment of an ontology that explains semantics and correlation of domain-specific vocabulary
(D2)	– Completeness/consistency
(D3)	– Structural extensibility

Organizational Model	
I/O	Used by all components for role abstraction
(OM1)	– Formal representation of corporate structure with respect to hierarchy, responsibility and specialization
(OM2)	– Role abstraction
(OM3)	– Coverage of official and unofficial roles
(OM4)	– Completeness/consistency/useful level of detail
Time Management Component	
I/O	Communicates temporal constraints with all other components
(TM1)	– Control and monitoring of temporal dependencies during enactment
(TM2)	– Synchronization with other components
(TM3)	– Integration of time constraints into workflow metamodel
Process Creation Engine	
I/O	Inputs an audit trail from the runtime engine and outputs process fragments
(PC1)	– Creation of instance-specific process fragments from an audit trail and general knowledge
(PC2)	– Goal-dependent invocation and creation of process fragments
(PC3)	– Robust and configurable input
(PC4)	– Metamodel-conformance of output
(PC5)	– Supplementation of output with a description
Runtime Engine	
I	Process models, interaction protocols
O	Audit trail, task assignment, synchronization with externals, monitoring
(RE1)	– Rights management/security
(RE2)	– Task assignment
(RE3)	– Instantiation of process models
(RE4)	– Schema evolution
(RE5)	– Late modeling/Case handling
(RE6)	– Flexible execution of instances (ad-hoc change, exceptions)
(RE7)	– Assure correctness/consistency of running instances
(RE8)	– Create an audit trail from events and incoming interaction protocols
(RE9)	– Allow annotations of events
Process Matching Engine	
I	Queries
O	Result set of matching data objects, eventually supplemented by a rating
(PM1)	– Queries contain quantitative/qualitative constraints and are supplemented by data objects
(PM2)	– Exact and approximate matching
(PM3)	– Descriptive and schema-matching search
(PM4)	– Syntactic, semantic or structural similarity matching
(PM5)	– Rated result sets with filtering threshold

Repository	
I/O	All data types with all other components
(R1)	– Temporary storage of data:
(R1a)	– Runtime engine state information
(R1b)	– Time management information
(R2)	– Permanent storage of data:
(R2a)	– Dictionary ontology
(R2b)	– Organizational model
(R2c)	– Process models with versioning information
(R2d)	– Archived process fragments
(R2e)	– Process compositions
(R2f)	– Audit trail
(R3)	– Efficient data representation
(R4)	– Basic access methods to stored information

Table 2.4.: Requirements summary

2.4. Process metamodel

A *model* in the context of workflow management reduces the complexity of systems in the real world in order to make it controllable[10]. By abstracting from reality, individual objects and relations of the real world are reduced to object types and relation types by filtering out irrelevant aspects of reality. The more detail is left, the more complex a model grows. Hence a process designer determines how much information is relevant and decides on the required level of complexity.

A *metamodel* defines a model for all models within a workflow management system. It establishes a formalism that defines the class of constructs which are allowed in models. Key dimensions[11] of metamodels are among others its granularity, control flow, data flow, organizational model, role binding and exception handling. Practically speaking, the metamodel determines the maximum expressive capability of all models built according to it. The metamodel both abstracts a "modeling language" from models and can be used to verify the correctness of models.

A *process metamodel* represents a process perspective view on a metamodel and shows only partial aspects of the total metamodel as it is used in a workflow management system. The following paragraph puts the process perspective into a bigger picture in order to give an idea of its classification.

[10] Compare [Müh96] p. 13 et sqq.
[11] Compare Lei and Sing [LS97] p. 3 et sqq.

Workflow perspectives Van der Aalst and Jablonski identify five different perspectives to characterize different aspects of a workflow management system [AJ00]. These perspectives are a good starting point to structure as shown in Figure 2.14:

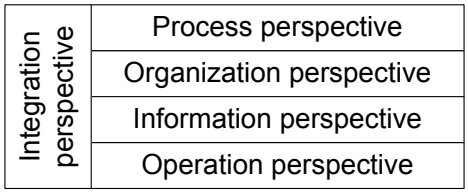


Figure 2.14.: Workflow perspectives (compare [AJ00] Figure 1)

The *process* perspective takes a task and control flow oriented point of view focusing on process definitions, their type and instantiation. The *organization* perspective focuses organizational structures characterized by roles, groups, responsibilities and their allocation. The *information* perspective is a data-centric view dealing with control and production data. Elementary operations performed by applications and resources form the *operational* perspective. They are used in the process perspective as elements for construction of data and control flow. The *integration* perspective finally links all views together.

This Section deals with the process metamodel of Emergent Workflow and therefore restricts its view to the process perspective. An overview of the most important components of Emergent Workflow's process metamodel is given in Figure 2.15. Further explanation on the shown elements will be given in the subsequent Sections. Notice that Figure 2.15 does not contain instance-specific elements such as instances, fragments or compositions to enhance readability.

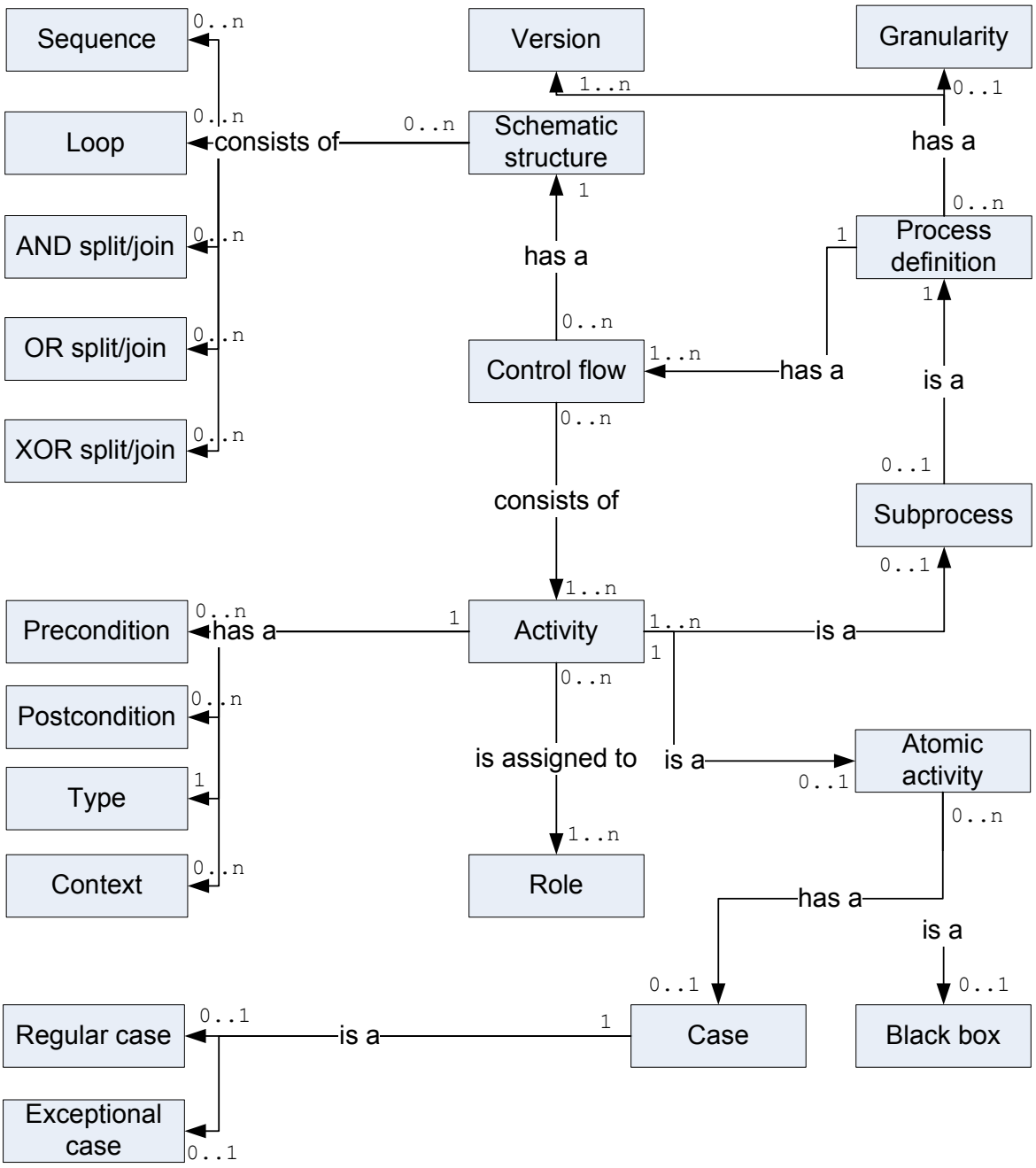


Figure 2.15.: Process metamodel

2.4.1. Process definitions

A *process definition* or process model represents the formalization of a business process. A business process consists of a manual part called the *manual definition* and an auto-

mated part named *workflow definition*[12]. The attribute "automated" is here used in a wider sense than addressing only processes which run without any manual interference. It rather refers to the set of processes which are supported by information technology. Workflow definitions consist of a number of items and relations expressing an automated process. These items are activities, resources and data objects. They are connected by structural relations which creating a control flow.

Granularity An issue with distinguished importance for Emergent Workflow is the definition, recognition and application of a process definition's *granularity*. It describes the abstraction level of an atomic or basic element within the process metamodel. Emergent Workflow aims at deriving process fragments from interactions and activities of active users. Users though have different perspectives, responsibilities and statuses within an organizational model. Thus, their perception of what an "elementary" task is differs significantly. Emergent Workflow requires the ability to cope with inputs that differ in level of detail and granularity.

Definition and recognition of granularity means to establish a common measure that allows the classification of all incoming fragments. That refers not only to the recognition of a top/bottom level task, but includes also quantitative measurement of intermediate level tasks.

A lower bound for the finest level of granularity is represented by the *stability of models*: A process model should be stable and not change on each instantiation due to persistent changes on the lowest granularity level. An event in a workflow management system represents the smallest recognizable element for an information system, however tasks outside the system may be even more detailed. Semantically, an elementary task should be chosen as the smallest stable and independent set of operations that form one logical unit. Being small is here characterized as a minimum amount of bound time and resources. The highest granularity level being the other end of the spectrum is the *top-level process*. It is basically a coarse view on the total process that does not allow any further abstraction with less details and a broader scope without loosing significant information.

Between these two extremes, intermediate levels of granularity exist. Their classification is most challenging due to the number of characterizations indicating a granularity level: First, the *hierarchical position* of the person who executed an instance is an indicator for the granularity of the underlying process definition. A task regarded as elementary by a project manager may represent a whole subprocess for the software tester subordinate to the manager. Second, the involvement of (eventually nested) *transactions* gives a hint on the abstraction level as transactions may be used both on higher or lower levels. The used *time* for the completion of one task as well as the *amount of resources* bound by a task classifies the individual granularity level of a reported process fragment.

[12] Compare the Workflow Management Coalition Reference Model [Hol95] p. 7 et sqq.

It appears very plausible that the total number of hierarchies (the "granularity of granularity levels") has been defined at some point in the workflow cycle, e.g. in the organizational model. This establishes an abstraction hierarchy with distinguished levels, defined by a number of quantitative, measurable characteristics. Then it is the job of the workflow management system to examine incoming fragments and classify them in a granularity level within the defined abstraction hierarchy using one of the characteristics mentioned. Especially with regard to semantic process fragment composition, this represents a fundamental step to enable meaningful composition.

Version One of the flexibility measures mentioned in the requirements of Emergent Workflow's runtime engine was schema evolution (see Section 2.3.7). It proposes that process models are adapted to a continuous changing environment by the application of change operations. Process instances that were instantiated on the same process definition have potentially differing process schemata. Consequently, a process type by itself does not clearly identify the structure of its instances or schema. Hence, a *versioning* of process definitions is proposed. An incrementing version number indicates a schema change and gives a clear reference to each version of a process definition.

Activity Within a process definition, activities are elementary functional units. Each activity consists of many different types of information whose composition allow its functionality. All parameters combined yield a case its identity. As mentioned before, Emergent Workflow does have requirements in terms of workflow flexibility and reusability. These requirements are reflected by the parameters specifying an activity as shown in Figure 2.16.

An activity performs a certain *task* and is identified by a *name*. An activity's character is specified by stateful *case* information which describes the case content, case attributes and conditions[13]. In Emergent Workflow, *case* information categorizes an activity and tells for example whether it is *regular* or *exceptional*. The number of case categories is extensible and they can be used to classify instance-based entries in an audit trail with respect to their relevance or likelihood to reoccur. As a further differentiated classification of cases may be useful depending on the application domain, the extensibility of this attribute is expressed by one or more *custom* cases. An activity *type* tells whether additional constraints have to be taken care of when an activity is processed. The regular case is an *atomic* activity. However, the activity can also be a placeholder for a *subprocess* or a *black box*. These impose specific execution restrictions to the activity, e.g. a black box (compare Section 2.3.7 on page 41) activity must be fully submodeled before the activity can be activated. One or multiple *descriptions* offer room to describe informally from one or multiple perspectives what an activity does. A *flexibility* parameter tells the workflow management system about the degree of flexibility of this activity. It can be either *fully* ad-hoc changeable, the change methods may be restricted

[13]Compare [AH02] p. 33 et sqq.

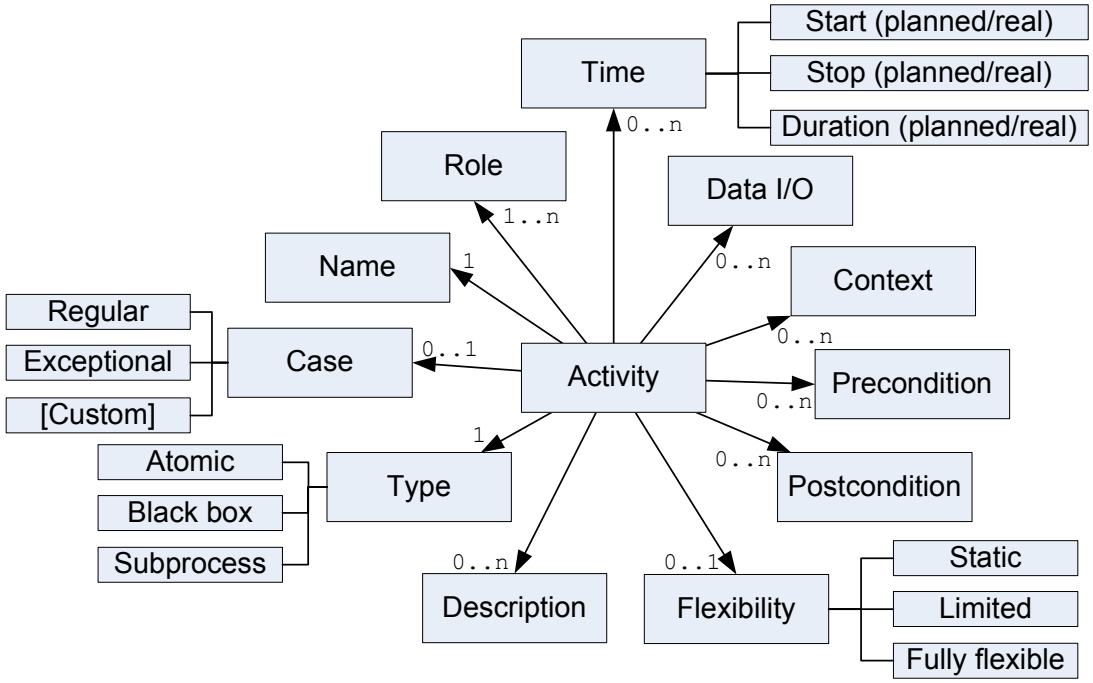


Figure 2.16.: Activity metamodel

to a *limited* type and change time (e.g. only description changes before activation) or the activity is totally *static* and does not allow any changes. Conditions split up into *preconditions* and *postconditions*: Preconditions tell about the conditions ought to be met before activation and postconditions guarantee a defined state after termination of the activity. *Contextual information* is a generic entry which captures relevant factors influencing activity processing or which are needed for post-hoc evaluation. *Data input and output* refers to data objects that are created, read or wrote during activity processing. *Time* entries hold temporal constraints for time management and are used to record the execution history of an activity instance. They contain start and stop times as well as a duration fields, each having a field for the planned and the real value. Finally, the *role* object identifies which organizational entities are allowed to process a definition.

Control flow A *control flow* interconnects activities being elementary parts into a continuous workflow. The structural elements of a control flow determine causal relationship of activities within a control flow. In order to classify the control flow expressiveness of workflows, van der Aalst et al. define *workflow patterns* [AHKB02]. A pattern abstracts from solutions given for concrete problems an makes more generic recommendations. By separating basic from more advanced language constructs, an incremental approach to the requirements on a modeling language is given.

The following enumeration gives a summary on basic patterns (No. 1 – 5) and selected advanced patterns (No. 6 – 9):

1. **Sequence**. This allows activities to be executed in sequential order. An activity is activated after its predecessor terminates.

2. **Parallel split**. The thread of control splits at a parallel split which is also called AND-split. A thread of control describes the path of execution which is headed by the currently executed activity. Multiple activities are activated after a common predecessor terminates.

3. **Synchronization**. The execution of multiple activities/threads of control is merged using a synchronization or AND-join. The next activity is activated as soon as all incoming parallel threads of control have arrived.

4. **Exclusive choice**. The thread of control has multiple choices to proceed on different paths. In contrast to a parallel split, only one of the available alternatives is exclusively chosen which is why this split is also called XOR-split. The choice is made upon control data or a condition.

5. **Simple merge**. As a counterpart to the exclusive choice, the simple merge or XOR-join activates the next activity as soon as one of the incoming path is activated.

6. **Arbitrary cycles**. This construct allows to execute one or more activities repeatedly. Control data or a condition check whether and how often a loop is passed.

7. **Implicit termination.** When no activity is currently active and none is available for activation, the process is terminated implicitly without reaching an explicitly defined terminal state.

8. **Interleaved parallel routing.** When in a set of activities, no two activities should be executed at the same time but the order does not matter, then interleaved parallel routing allows the activation as an unordered sequence. That is, each activity is executed only once in a nondeterministic order.

9. **Milestone.** The activation of a certain activity depends on whether a milestone has been reached without expiring. The milestone being a condition is defined as a specific compound state of multiple activities of the control flow.

Patterns 1 through 5 are essential for even very basic structured workflows. For more advanced processes, pattern 6 is useful but complicates execution considerably: The introduction of a loop means that activities can be activated multiple times per execution and their state has to reset when a new loop iteration is started. Analysis becomes harder and due to the possibility of infinite loops, correctness can hardly be guaranteed. Patterns 7 to 9 are not required but "nice-to-have's" as they support goals of Emergent Workflow: Implicit termination facilitates handling and correctness checking of incomplete process models or late modeled subprocesses. Interleaved parallel routing increases flexibility for many scenarios where the order of activities does not matter:

In fact, any other construct would impose artificial and unnecessary regulation on such cases. Finally, the idea of having a milestone element for control flow greatly aligns with running a quality-gate driven process such as the V-Model (see Figure 1.4 on page 8).

Notice that these patterns do not take into account flexibility-specific control constructs for the integration of subprocesses, exception handling and case-handling. Subprocesses need a hierarchical integration which enacts an independent process definition upon the activation of the activity representing a subprocess. For exception handling, an explicit description of an exception handling routine and a description on how to jump back and forth to the routine are desirable. A dedicated version of an XOR-choice/merge supplemented with implicit termination functionality would be a starting point for a structural control element giving dedicated support to exception handling.

The question on which workflow patterns to integrate in Emergent Workflow's process control structure has conflicting goals in its background: High functionality and flexibility can be offered by a complex process model with manifold control structures. But it also brings along much more complicated construction, changeability and correctness checking than a simple process model. Moreover, an easy-to-use process model is more likely to be understood and accepted by most workflow users. If a highly functional model is chosen, then an interface must be built around the models which either provides strong support or hides the system's complexity by translating the complex internal model to a more simplistic outside view and vice versa.

2.4.2. Process instance

A process instance puts the specifications of a process definition into practice. Each process definition can have multiple instantiations, but each instance refers to only one definition. An instance is an independent object residing inside the runtime engine during enactment. In the context of a process instance, each activity is instantiated and associated roles are resolved. That is, an activity is assigned with one ore more individuals in the organization who occupy an according role. How and when resolution takes place is decided by the runtime engine's role resolution policy. During execution, a process instance occupies resources such as the individuals executing it, use data or server-side applications. Beyond that, it traverses a number of states between its instantiation and termination. If from all possible states a terminal state is reachable, a process instance is called *correct*.

Process and activity state model A process instance has usually a defined start and end state and traverses a number of intermediate states from the one to the other. An initial state transition is mostly performed after initialization is completed by activating the first activity. The overall state of a process instance is determined by the set of states of its activities. An exemplary activity instance state model is shown in Figure 2.17.

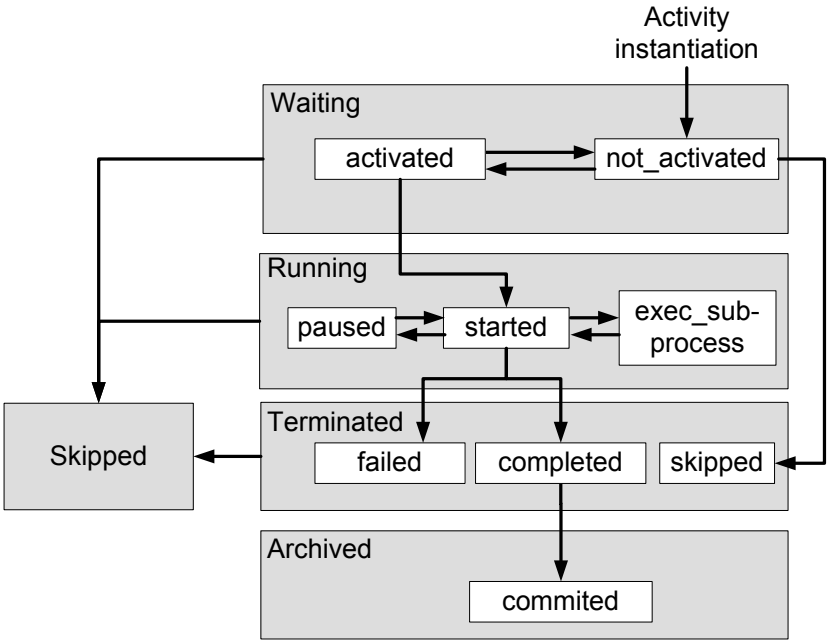


Figure 2.17.: Activity instance state model (adapted from [Rei00] Figure 4-1)

Figure 2.17 shows a coarse state model which applies for both, an activity and the overall instance as shaded boxes. Within the shaded boxes, white boxes represent more detailed activity states. Arrows between shaded boxes and between white boxes indicated state transitions of the instance and the activity respectively. An instance switches in an initial *waiting* state as soon as its first activity is instantiated. It moves from there into a *running* state as soon as the first activity is running and *terminates* as soon as the last activity has either failed, completed or was skipped. Upon successful termination, it is being *archived*. From any of the first three states, the instance can be *skipped* which means its execution is aborted at some point.

An instance starts in a state *not_activated* and gets *activated* when the thread of control arrives. From there it transits into a *started* state as soon as all of its resources are allocated. While it is running, it can switch back and forth to the sub-states *paused* or to *exec_subprocess* in case a subprocess is associated. Upon termination, an activity can either *fail*, e.g. due to errors during execution or *complete* successfully. The last step after completion is a *commit* to the archived instance. As multiple paths of control flow through a process schema exist, an activity can possibly be never activated. In that case it transits straight form *not_activated* to *skipped*.

As already indicated by the final archiving state, a history of state transitions is recorded and saved as part of the audit trail. This is a requirement due to the demand of Emergent Workflow for the analysis and reuse of part instance executions.

The flexibility requirement of Emergent Workflow (see Requirement summary in Section 2.3.10, (RE5)) to enact incomplete instances has further impact on the metamodel of

process instances. Black boxes represent missing subprocesses and are modeled during run time. Also ad-hoc changes (RE6) interfere with instance execution. Both cases modify the instance's execution state and after each modification, consistency checks are mandatory to assure the *legality* of a state. If for example a non-activated activity is inserted in an area which precedes activities that have already terminated, it becomes impossible for the instance to terminate correctly. If an instance modification leads to an illegal state, either the user or the runtime engine has to care for its correction (RE7).

Furthermore, each instance should archive any extra *annotations* which were entered by a workflow user during enactment (RE9). This allows him to supplement the the archived instance with useful, informal notes for later reuse.

Instance fragments When an audit trail is examined by the process creation engine, a partial process instance is derived and is referred to as an *instance fragment* (PC1). It links to the archived execution history and contains thereby references to executing individuals, occupied resources, timing informations and results from instance flexibility measures.

From a system perspective, an instance fragment is a portion of an archived process instance put into a formalism according to the process metamodel. Compared to a process instance, specific start and end states are missing and the correctness of the schema is unchecked. Notice that it is built from the audit trail which has a certain level of detail according to the originator of the trail. That is, each fragment has an individual *level of granularity* describing its richness of details. An explicit representation of granularity is fundamental for further reuse of instance fragments as the next Section reinforces.

2.4.3. Process compositions

A complete workflow transforms an initial business requirement (the precondition) into a state that realizes the business goal (the postcondition). It does so by a number of steps implicitly traversing a state-space. The *composition* of fragments equals to the identification of a sequence of tasks that transforms a precondition state into a state complying with the final postcondition for an "overall" instance. In fact, composition is performed *vertically* and *horizontally*: Vertical composition of fragments represents the alignment of different views and hierarchies related to a common part of the overall process. *Horizontal* integration links causally related process parts sequentially together.

The overall instance is identified by a business requirement such as an order number in a production process. From an employee's point of view, the overall instance is invisible and appears only as smaller instances of lower-level tasks. The reassembly of those fragments reveals the structure of the virtual overall process.

Theoretically it would be possible to combine fragments of different instance audit trails.

In fact it could be very much easier to assemble all parts of an overall process from multiple instances. However, their composition can be difficult and the usefulness of its outcomes is questionable. Because of the flexibility measures offered by Emergent Workflow, different instances are individually able to tailor a process by modifying instances, schemas and applying cases. When combining different cases or schema versions, they may not only collide syntactically but do not match semantically either: A cross-instance composed overall process starts with an business requirement and ends with a different business goal which does not reflect reality. That is why only compositions consisting of instance fragments belonging to one overall instance are meaningful.

Relation types Technically, a composition is a *relation* defined on a set of process fragments. The involved types of relations are shown in Figure 2.18.

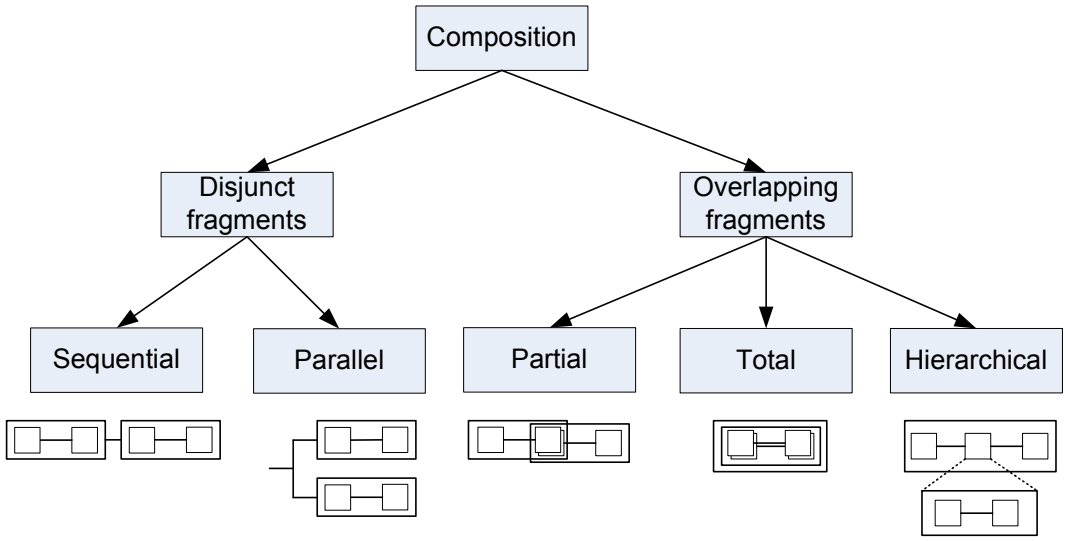


Figure 2.18.: Types of relations in fragment composition

Figure 2.18 makes a distinction between *disjunct* and *overlapping* fragment relations. A *sequential* relation directs the control flow after the termination of the first fragment to the second fragment. A *parallel* relation splits the thread of control and directs it to both fragments at the same time. Overlapping relations are more complicated as they correlate activities within fragments. A *partially* overlapping relation correlates a subset of both fragments with each other. If each activity of one fragment is correlated with an activity in the other fragment, the overlapping is *total*. A special case of overlapping relations is a *hierarchical* relation. It depicts subprocess relations by correlating one activity with the complete second fragment. When the thread of control arrives at this activity, the activity pauses and forwards the thread of control to the lower-level fragment. As soon as the thread of control returns, the activity is terminated and the top-level fragment continues its execution.

Notice that correlating an activity means the creation of a compound activity merged of the attributes specifying each activity (see Figure 2.16 on page 58). This is either achieved by adding both attributes (e.g. all roles from both activities are added to the new activity) or choosing one of them (e.g. the choice of a flexibility level) When creating an overlapping relation, either the system or the composer must take care of merging each overlapping activity. Each mentioned relation relates two fragments, hence for the composition of an overall instance, multiple relations must be applied. That requires a composition to conform with the same requirements as a process fragment, that is, from an outside look it behaves and looks like a process fragment.

Supportive constructs One of the key enablers of fragment composition is an explicitly expressed *level of granularity* on each fragment and composition. Any automated support for finding matching fragments is based on a quantitative metric to assess similarity, one of which is granularity. Besides other attributes such as roles and temporal information, it helps to determine which fragments can be composed in a semantically meaningful way and where a hierarchical relation should be applied.

As it is likely that fragments do not match each other perfectly or parts of the overall process are missing, elementary tools help to interconnect a control flow with gaps. These are constructs like a *spontaneous transition* to interconnect activities or an *empty fragment* in order to easily integrated a hierarchical relation into a composition. Furthermore, a *black box fragment* can indicate missing parts. In case the composition gets enacted, this construct fills up gaps which were not covered during composition and behaves similarly to a late modeling instance in the runtime engine.

Fields of use Process composition is performed by process designers who use designated client applications for modeling support. An example on how relations between fragments could be detected is given in the example below.

Example 16. Assume an activity "window power lifter mounted" with the context "assembly left front door" is in one fragment, activity "side window lifting motor mounted" with the context "assembly left front door" is in another fragment. By doing a dictionary lookup, a process designer can find out that the side window lifting motor is a part of the window power lifter assembly unit (see Figure 2.10 on page 22). From that information he can conclude, that the corresponding activity "mounting window lifter motor" is a sub-activity of the activity "mounting window lifter". If further examination of the temporal execution history, the involved roles and further context information (such as a serial number on the body) confirms that assumption, very likely a subprocess relation has been found.

As a composition tool is considered as a special case of user interface and the fields of use were not mentioned in Section 2.3.1, a short paragraph on the fields of use for compositions is added here. Process compositions are created by process designers for

in-depth *analysis*. By constructing a composition, designers receive a big scale view on the course of execution of an instance. While workflow participants are most able to optimize a partial process on a small scale due to their knowledge and experience, process designers use a composition as a tool to understand and optimize big scale dependencies of an overall process. Such analysis can be supplemented by a *simulation* of past executions using the continuous documentation given in a composition. Annotations made by workflow users are attached to fragments and accumulated on them is given in a composition. They represent an excellent summary on *lessons learned* during the execution of an instance. Drawbacks and conclusions lead to improvements in the big scale process and can be applied for example to the V model (see Figure 1.4 on page 8).

3. Related approaches

This Chapter presents a number of approaches which are related to aspects of Emergent Workflow. Each Section introduces at the beginning the fundamentals of the underlying field of research. By summarizing selected projects, interesting aspects of mostly recent work are highlighted. Each Section closes with an assessment of usefulness of the presented concepts in the light of Emergent Workflow.

3.1. Case-based reasoning

3.1.1. Fundamentals

Case-based reasoning (CBR) is a methodology that can be used to enhance flexibility in process management. It builds fundamentally on the hypothesis that reasoning is reminding of useful information. The origin of this automated learning approach lies in *Artificial Intelligence* research.

As introduced by Aamodt and Plaza [AP94], the idea of CBR is to solve problems by using knowledge gained by previous experiences which are referred to as *cases*. Because each solved case is added to a *case base*, it extends incrementally the available experience within a problem domain. CBR is a *learning* technique because the knowledge about the problem grows independently from the reasoning method and fosters better or easier finding of solutions.

Commonly the term CBR is used in a wider sense and refers to various reasoning methods. Strictly speaking it differs however from other reasoning types. Those varying aspects include methods for retrieval, management and utilization of past cases and general domain knowledge as well as matching and adaption procedures. A list of related reasoning methods is given below:

- **Exemplar-based reasoning** Here a *concept* is defined as the set of its exemplars. Solving a case in this scope denotes a *classification task* where the matching class of problem is found. As each class represents one single solution for a particular type of problem, the class that shows most similarities is chosen as a solution. A concept definition is learned when an unclassified problem can be classified correctly.
- **Instance-based reasoning** This is a specialization of exemplar-based reasoning that aims at *automated learning* without user interference and focuses on a *syn-*

tax-oriented approach. Less background information is available, the data model is relatively simple and a bigger number of cases is necessary to find a concept definition.

- **Memory-based reasoning** The case base is seen here as a large piece of memory. The reasoning procedure corresponds to navigating and searching through the the memory. Consequently, herein types of organizing and accessing the memory and processing methods are focused.

- **Case-based reasoning** Although the term *case-based reasoning* is used more generic throughout the thesis, it differs typically from the other reasoning methods mentioned in a number of aspects. First, a case is considered rich of information and has a rather complex organization in contrast to the data model of instance-based reasoning. Second, more general background knowledge can be used in a situation-dependent context. Finally, CBR distinguishes itself by the ability to *modify* a retrieved solution, which allows and implies user interference.

- **Analogy-based reasoning** Although closely related to CBR, *analogy-based reasoning* focuses on finding *analogies* between problem domains. That ability characterizes methods which solve new problems by basing their solutions on solved problems of *different domains*, whereas CBR matches cases within one problem domain.

Process model view on CBR Effective problem solving with CBR involves a number of steps. When a new case comes up, it must be first analyzed to determine the type of problem. Next the case based can be searched for similar cases that match the new problem sufficiently with respect to chosen criteria. If a previous case matches the new case, it is used as a proposal for a new solution. After eventually necessary adaptions have been made to the proposal and it has been accepted as a solution for the new case, it can be added to the case base and becomes a *learned case*. From this point on, the solution for the next new case relies on the improved case base. Formally, these actions are represented by *retrieval*, *reuse*, *revision* and *retainment* phases. The cyclic nature of this procedure becomes evident by a glance at Figure 3.1 which illustrates the steps mentioned in a generic CBR cycle.

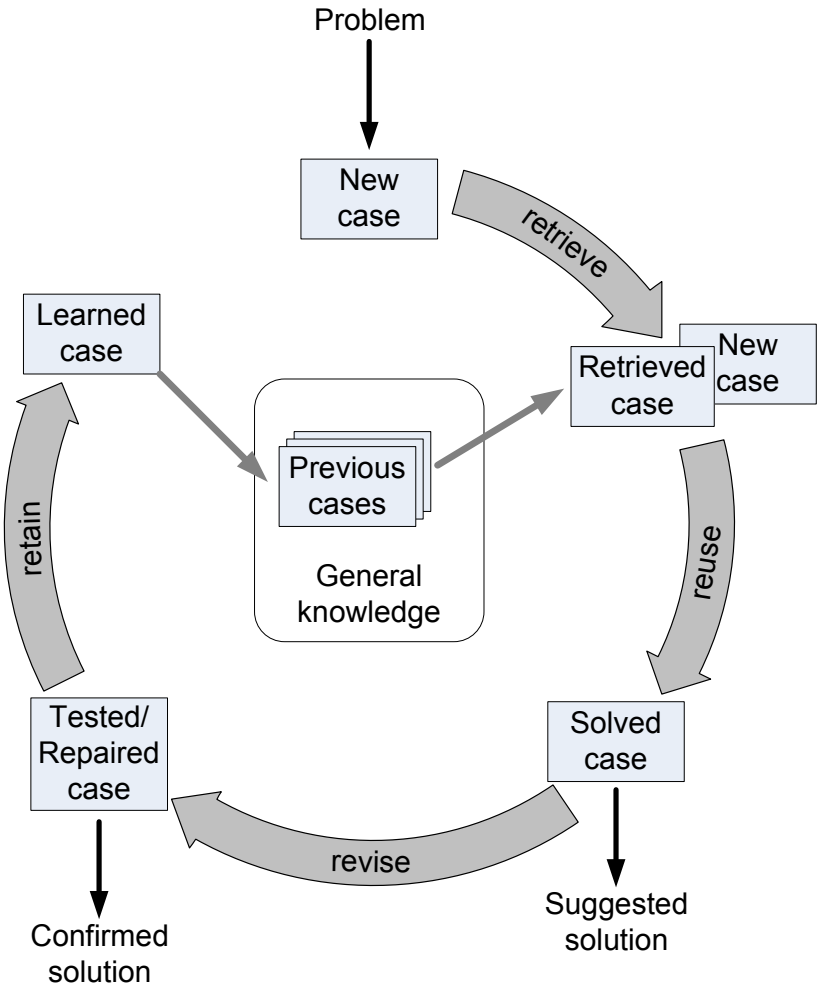


Figure 3.1.: CBR cycle (compare [AP94] Figure 1)

The generic CBR cycle in Figure 3.1 consists of the CBR main tasks: Upon the reception of an incoming new case, the user retrieves the most similar cases by using general domain-specific knowledge and provided case retrieval methods. He further reuses the available information to solve the new case, revises the proposed solution and retains interesting and relevant information in a learned case for future cases.

As the illustrated CBR cycle already indicates, the main problem areas of CBR are knowledge representation and methods for retrieval, reuse, revision and retainment. Within each area, one faces a number of questions whose architectural answers affect the functionality of a CBR implementation.

Knowledge representation Within a case base, gained experience and learned lessons are stored. Together with generic domain knowledge it is fundamental for the overall problem solving process. Thus, it is crucial to decide on a data structure that is both an effective knowledge representation and efficiently accessible. What information should

be stored in a case? The more information is packed into it, the more likely it is to detect commonalities between related cases. Much irrelevant or redundant information however makes the case base hard to use and reduces efficiency. The need for efficient structures does not only apply to the organization of cases, but also to the internal structure of each case. A chosen data structure needs to be extensible as the case base incrementally grows with each learned case. The more indexes and other data structures are created for accelerated output, the more administrative data has to be updated for each alteration of the case base. Finally general domain knowledge has to be integrated with the case base in a way that integrates them in requests. It may have for example the form of a rule base containing "best-practice" rules which are applied to each query before it is directed to the case base.

Case retrieval is clearly identified by its input and output: It starts with the reception of an incomplete problem description as input and outputs the best matching case from the case base. Three major steps lead to the desired outcomes. The input is first analyzed in order to *identify its features*. This corresponds to the acquisition of a true understanding for the present problem. The set of features is used for an *initial matching procedure* in order to identify a number of candidates within the case base that are potential solutions. Next a *selection process* of the most promising results refines the matching set until a best matching case becomes evident. Eventually matching and selection are one single step, but they usually differ from each other by the applied depth of analysis. While matching is more superficial, selection analyzes more detailed the relevance of identical and non-identical features.

Case retrieval needs a measure or metric to compare the similarity of cases and the relevance of features: Those measures can be either based on *syntactic* or *semantic* similarities. Syntactic measures are simpler to apply and return a rather superficial result while semantic similarities are more accurate and more complex to obtain. Semantic matching requires general domain knowledge in order to interpret for example contextual information. Identifying a set of features from the given input and concluding on a problem type requires a type of a semantic network that correlates terminology.

Example 17. A straightforward example for a similarity metric is used in CBRFlow [WWB04]. A query Q on the case base represents a new problem and is matched against a solved problem C from the case base for similarity. Features of cases are detected in this approach by a question-answer process; thus a set of answered questions $\{QA_1, \ldots, QA_m\}$ comes with Q. A comparison of these questions and answers in pairs yields an observation whether the pair is matching or not. The similarity is calculated as the normalized difference between the number of shared observations and the number of conflicting observations.

$$\mathrm{sim}(Q, C) = \frac{\mathrm{same}(Q_{QA}, C_{QA}) - \mathrm{diff}(Q_{QA}, C_{QA})}{|C_{QA}|}$$

Case reuse is based on the identification of matching and differing attributes between

the old and new case. While the useful parts of the old case have to be extracted into the new case, the non-matching parts are to be left out. In a more simplistic scenario it is sufficient to make a comparison for similarities – the differences between the cases appear irrelevant and are abstracted away. Then both cases are reduced to the problem class and the retrieved case is *copied* as a solution to the new case. More realistic however is a scenario where the retrieved case can not be transferred immediately to the new case but requires *adaption*.

Adaption is achieved by either finding a *transformation* that translates the old solution into a new solution or *deriving* the past methods such that it produces a solution for the new case. Transforming the solution is only appropriate though if the case is rather output oriented and the procedures themselves are not crucial to the success of a case.

Example 18. During the process of designing a car body, mechanical engineers use a virtual prototype called *digital mock-up* which is equipped with methods to check for collisions of body parts during development. Dependent on the type and severeness of collisions, synchronization of collaborate work and resolution of problems can be classified in several cases. For minor issues the transformation of older solutions is likely to be sufficient because here only the outcome (which is a resolved collision) is relevant. However, if the collision is more complicated and involves meetings of several disciplines, then the resolution process itself in the form of inter-personal communication is important as well and is influenced by many external parameters. Comparable cases from the case base must then be adapted and reenacted instead of a replication and modification of their former solution.

Case revision evolves out the lack of correctness or completeness of a reused solution. It includes the evaluation of the reused case in order to clarify its deviation from current requirements. After this has been found out by simulation or applying domain knowledge, a learning effect is accomplished by extending the case base with the new findings. Furthermore, faults in the reused case may be repaired by generating explanations for them. Based on an explanation, modifications can be developed to repair a case solution. After a case has been revised, it should be assured that it can be applied without exceptional behavior.

Case retainment enables the learning procedure within CBR by incrementally extending its case base. Depending on whether the new case has been derived from a past case or was newly defined, existing cases are generalized by supplemental features or new cases are added. Problem and solution descriptors as well as indexes for case retrieval methods have to be refreshed for the updated case base to take effect.

3.1.2. Applications

CODAW The Case-Oriented Design Assistant for Workflow Modeling (CODAW) is an approach that aims at supporting workflow model reuse during workflow design.

Madhusudan et al. point out the lack of useful standards regarding process model storage, retrieval, reuse and assembly. In [MZ03, MZM04], they present an architectural proposal for *case representation*, *case retrieval* and *case composition*.

Manual process modeling is here considered a traversal of a "design space" defined by a large number of process model alternatives and the selection of an optimal process model that reflects the given problem best. Two phases are identified within process design:

In the first phase, relevant business tasks are put into a *partial ordering* that satisfies all preconditions and postconditions. Multiple process models can be found that meet these requirements. In the second phase, the favored process model is *selected* and *completed*: Routing is optimized with respect to flexibility and parallelity and appropriate agents and resources are associated with the model. In both phases of design, process knowledge from the repository in the form of cases may be reused.

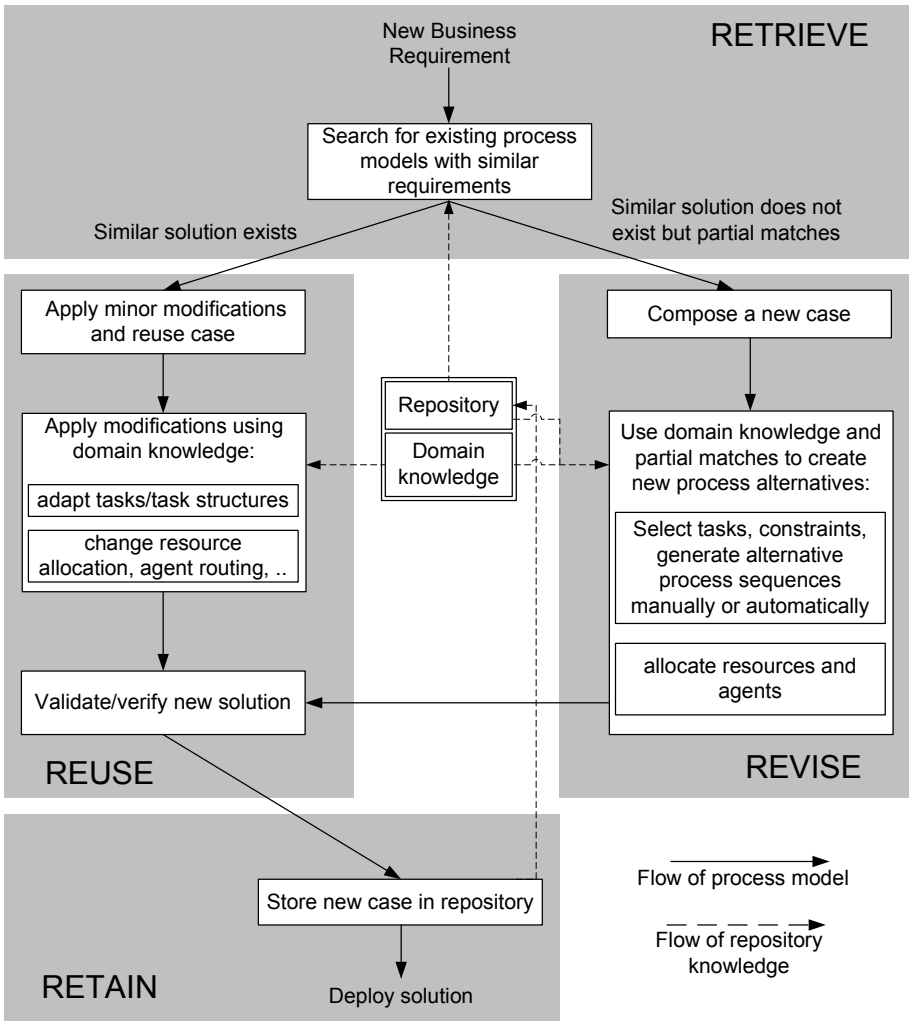


Figure 3.2.: CODAW workflow design process (adapted from [MZ03] Figure 2)

Figure 3.2 shows the simplified design process of process modeling using CODAW. In compliance with the generic CBR cycle (Figure 3.1 on page 68), its procedure splits up into four phases: Retrieval, reuse, revision and retainment. An incoming business requirement initiates *retrieval* by searching the repository for process models with similar requirements. If a match is made, then the found case is slightly *modified* and *reused*, otherwise a new process must be composed. The modification for reuse are minor structural/semantic changes such as the replacement of a task or the modification of the process schema. Additionally, instance-specific settings such as resource allocation need to be set up individually, even for processes of the same type. Validation and verification uses measures such as domain-specific correctness checks, visualization or simulation and assures the correctness of a reproduced case. For knowledge *retainment*, the newly developed process is not only deployed to the workflow management system after checking, but also stored into the repository for later reuse. If no suitable template could be found during case retrieval, a new case has to be composed. Creation relies on domain knowledge and eventual partial matches for the synthesis of new process alternatives. This is done either entirely manual or is supported by a planning software, whose basic principles will be mentioned below. Revision is finalized in the same way by validation and verification, retainment and the solution is deployment just like a reused case.

Case representation is here approached by separating *prototypical cases* from *instance-level cases*. In the terminology of this thesis, a prototypical case comes close to what we refer to as a process definition and an instance-level case would then be a process instance. Prototypical cases contain the sequence of activities and represent a process schema for a generic business requirement. Instance-level cases depict the execution trail of a prototypical case for a specific input. One prototypical case can be associated with several instance-level cases.

In CODAW's process ontology, the existence of primitive tasks which can be combined into more complex processes is assumed. A process schema defines then internal structure of a composite task. Furthermore, it is possible to create hierarchical structures by reusing a schema as a component tasks in another process.

The implementation of process definitions and instances is based on XML Schema and leans towards standards such as XPDL[1], WSFL[2] , XLANG[3] and BPEL[4]. A repository organizes prototypical and instance-level cases as well as a collection of primitive tasks. Cases are arranged in a hierarchical directory structure in flat XML files. It indexes cases according to their functional application area, task and organizational structure.

[1]See the XML Process Definition Language Specification Version 1.0 Final Draft: http://www.wfmc.org/standards/docs/TC-1025_10_xpdl_102502.pdf

[2]See the Web Services Flow Language Version 1.0: http://www4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf

[3]See the XLANG Initial Public Draft Release http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm

[4]See the Web Services Business Process Execution Language Version 2.0: http://www.oasis-open.org/committees/download.php/11600/wsbpel-specification-draft-022705.htm

Retrieval is implemented by plain text search or XQuery requests doing exact matches on defined XML tags.

An example of the XML representation of a new product development process schema is given in Appendix A.1.1. It becomes evident by the example that a process schema consists of three sets of tags (in parentheses the corresponding line numbers of the example is given): General *descriptive* tags for ID, name, type and description of the schema (lines 2 – 5), *workflow level structural* tags to describe the overall process (lines 6 – 14) and *task level* tags specifying tasks and their parameters (lines 15 – 64). It is substantial to notice that each task is defined in three ways: First, a state-space declarative AI planning based representation is given by the tag `TaskDesign` (lines 21 – 37) and is used for case composition. Second a formal model representation using process algebraic representation is denoted by tag `TaskFormal` (lines 38 – 40). In the example, this is textual description of a simple sequential finite state automaton. Third, a procedural task definition denoted by `TaskDefn` (lines 41 – 63) highlights the implementation of related attributes of the task such as roles (lines 42 – 48, named "Agent"), procedure description (lines 49 – 53) and data I/O (lines 54 – 62).

An illustration of the XML schema of an instance representation is shown in Figure A.1 in Appendix A.1.2. IDs identify the instance and the process schema it originates from. It captures its execution history by recording data inputs and outputs, execution performance metrics and a history of events.

Similarity-based case retrieval embodies a flexible notion of similarity that combines features of domain knowledge and process graph structures. Case retrieval in CODAW relies on an algorithm by Melnik et al. named *Similarity Flooding Schema Matching Algorithm* [MGMR02]. It is an inexact and generic similarity based schema matching algorithm. The strength of this approach lies in its versatility that makes it applicable to many structures, including both process schemata and instances.

It takes two graphs as input and generates a mapping between nodes that appear to correspond. A threshold is defined in order to filter out the most relevant matches. Similarity Flooding is meant to support a human in the matching process rather than creating a complete matching autonomously. That is why the final step is to present the most relevant matches to a human who revises and corrects them where needed.

The algorithm can be briefly described as follows: First it transforms the two input structures into directed labeled graphs. The core idea of the matching procedure between those graphs is to combine two ideas: As a starting point, a string comparison of common prefixes and suffixes between designations of graph elements is performed. Based on the assumption that, if two nodes are similar, then their neighboring nodes are similar as well, an iterative fixpoint computation follows. Thus, a node propagates its similarity to its adjacent nodes, who themselves continue propagation of similarity updates until a stable state, the fixpoint, has been reached. Propagation relies on flooding algorithms

(such as the Distance Vector Multicast Routing Protocol[5]) which explains the naming of Similarity Flooding. Notice that this algorithm does not have any semantic knowledge about the contents of the input it is processing. Instead, semantic knowledge about process models is replaced by the explicit representation and mutual influence of name, type and attributes of process model elements.

Similarity Flooding for workflows runs on a graph representation that explicitly models nodes for tasks and control structures such as in a Petri Net. While task nodes are named according to their underlying task type, all control nodes share a common descriptor in order to make their common type recognizable for Similarity Flooding. By defining the similarity measure between any control and task node as 0, an accidental match between different element types is avoided. While similarity values applying for matches of control nodes are limited to discrete values 0 or 1 (representing no match or full match respectively), matches of task nodes can range continuously between 0 and 1 according to their substring similarity.

The matching algorithm results in a ranked list of map pairs including their final similarity after convergence, from which the most relevant subset can be chosen as a final mapping. As a manual selection of this map would turn out to be a tedious task, a filtering process is applied before the resulting table is being presented to the user. Filtering rules rely on experimentally determined similarity thresholds or can make use of domain specific knowledge. The accuracy and efficiency of Similarity Flooding for workflows is mainly determined by a good choice of threshold and is further examined experimentally as found in [MGR02].

Functional limitations apply to Similarity Flooding as it only works and performs on directed graphs sufficiently. Based on its algorithmic idea, it accounts only for semantic similarity which is being reflected by node and edge labels and topological similarity of the compared graphs. As far as computational resources are concerned, the size of input graph nodes is limited by the fact that intermediate graph structures are based on cross product operations with exponential costs.

Case composition is invoked by CODAW if either case retrieval fails to find a matching recent case or when retrieved cases have to be composed. If no matching case was found, a new process model is generated by the composition of primitive tasks. The input of case composition describes a business problem defined in a planning language. The output is a set of declarative process models that characterize its attributes and constraints in an enumerative style.

Case-based planning in the CODAW framework uses the *Simple Hierarchical Ordered Planning* (SHOP) algorithm, an implementation of the Hierarchical Task Network planning technique. Its approach is to create plans by task decomposition and constraint satisfaction. The SHOP algorithm supports reasoning on interactions between task preconditions and postconditions during state-space search for developing plans. Addi-

[5]Defined in RFC 1075, compare also Kurose and Ross [KR93] p.308 et sqq.

tionally, SHOP allows reuse of appropriate prototypical and instance-level cases from repositories during problem solving.

The SHOP algorithm works roughly as follows: A planning problem is first specified by an initial task network, which is a collection of tasks that need to be performed under a specified set of constraints. The planning process decomposes tasks in the initial task network into progressively smaller subtasks until the task network contains only primitive tasks or operators. The decomposition of a task into subtasks is performed using a method from a domain description. This method specifies how to decompose the task into a set of subtasks. Each method is associated with various constraints that limit the applicability of the method to certain conditions and define the relations between the subtasks of the method. The planning algorithm performs a recursive search of the planning state space via task decomposition and constraint satisfaction. It terminates either when all a solution has been found that meets all pre- and postconditions or when it tries to backtrack a composite task that does not offer any more methods to decompose successfully.

The SHOP algorithm is able to generate a sequential workflow which requires post-processing in order to add concurrently executed tasks by the analysis of data and causal dependencies. The effectiveness of the SHOP algorithm strongly depends on an appropriate design of the predicates, operators and methods. In the worst case, it explores the complete search space incurring exponential costs. With respect to that issues, time-out mechanisms may be used to ensure termination.

Conversational case-based reasoning A *conversational case-based reasoning* (CCBR) system as proposed by Weber et al. in their system CBRFlow [WWB04] is a hybrid reasoning approach combined with user interaction. *Rule-based reasoning* procedures are supplemented by *case-based reasoning* which improves adaptivity of the overall system. Business rules are a set of statements representing general, domain-specific knowledge that regulate the course of processes, such as instantiation or exception handling. A set of business rules is predefined in the process model and is annotated during run time with cases having context-specific information. The business rule set defines the default system, whereas cases add specific knowledge gained by previous concrete problem situations.

CCBR approaches to integrate machine learning methods from CBR with continuous user interaction in order to enhance the learning process and to overcome weaknesses of pure machine approaches. A case in CBRFlow consists of a free textual *description*, a set of question-answer pairs give the *reason* for the case and *actions*. An action is specified by the change operations taken and the subject they are operating on. In contrast to traditional CBR, problems do not need to be specified a priori completely in CCBR. Instead, the system is assisted by user interaction in an initiated dialogue of questions and answers that helps to retrieve the desired case or to evaluate the relevance of case features. This dialogue proceeds incrementally until the user has pinpointed a solution.

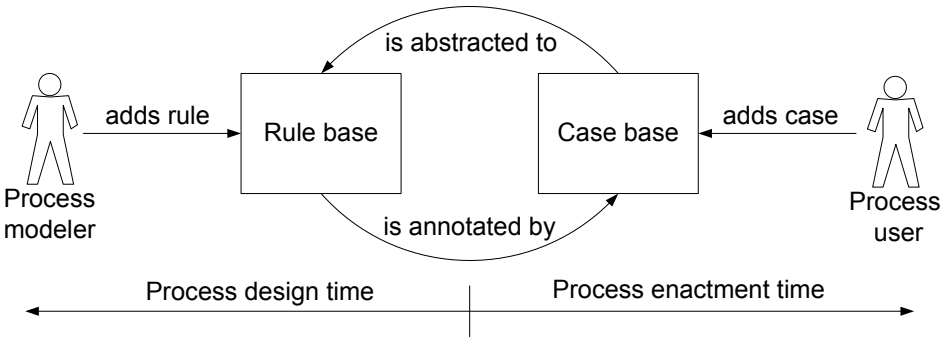


Figure 3.3.: Adaptive workflow management approach with CCBR (compare [WWB04] Figure 1)

The adaptive workflow management approach of CCBR as shown in Figure 3.3 starts with an initial model. It consists of a set of business rules which are formalized for example in event-condition-action (ECA) form. These rules describe the model's control flow. During run time, instances of a model are created and users work with them. Whenever changing requirements or exceptions cause deviations from the model, a user annotates the corresponding business rule by adding a case. The case describes, in which context what kind of deviation between reality and model occurred. Over time a case base becomes established and shows, which kinds of exceptions from business rules occur more frequently than others. Based on that information, process modelers analyze those cases which describe concrete exceptional situations and abstract from them changes on business rules that incorporate the modified requirements. Thus the original process model is adapted to changing environments. As changes to case and rule base can be temporally overlapping or take place concurrently, a process model adapts incrementally without strictly separated design and enactment phases.

The fundamental distinction between rules and cases is that rules are applied automatically, whereas cases require user interaction. Reasons that induce cases such as exceptions or helpful annotations are too manifold than to be processed automatically and need to be checked manually. Cases are used in two ways: One type supplements the rule base by referring to specific contexts that cause changes. Alternatively a case updates a specific rule by "hardcoding" the case with it. That way a case can update a particular rule.

Applying hybrid reasoning in the form of CCBR bears several benefits over pure rule-based reasoning: Explicit initial process modeling is allowed to be incomplete or rather low-detailed. Especially before process enactment it is very hard or expensive to find all important rules or parameters influencing the process. As improvement of models with CCBR is more continuous than reengineering the rule base, a starting model is more flexibly adapted to the starting and continuously changing environment. Cases enable users to express immediate manual adaptions and serve as a decision support system. The selective transfer of cases into rules differentiates one-time exceptions from

more systematic changes caused by new circumstances. Due to its high frequency, the latter is abstracted into the model and becomes more efficient because it is executed fully automated.

WorkBrain *WorkBrain* is a system presented by Wargitsch et al. in [WWT97, WWT98] that integrates an *evolutionary* workflow management system with an *Organizational Memory Information System* (OMIS). A workflow management system being evolutionary addresses here a step-by-step development of the workflow that involves its participants in incremental process design.

The concept of an *Organizational Memory* represents an organization's ability to retain previously made experience, so-called "learned lessons". Its existence is desirable as it leads to improved performance and higher effectiveness. Usually it does not exist implicitly within an organization's structure but has to be established explicitly. An OMIS tries to implement the concept of an organizational memory with information technology. On its own, an OMIS does not differ substantially from common information systems or databases that span over a large knowledge domain. Therefore, it does suffer the same weaknesses because its access methods are often insufficient to find data and to interpret them correctly: Usability is hard to manage for potentially many types of involved users, the costs for maintenance in order to keep the data basis up-to-date, consistent and correct are high.

By integrating an OMIS with a workflow management system, WorkBrain attempts to overcome each individual system's deficiencies[6]. In order to put its evolutionary idea into practice, its concrete learning approach is twofold: *Learning by example* enables workflow users to introduce spontaneously new elements into the process. This provides a more creative way of design and positions its results closer to operations. At the same time, process designers observe and reflect those changes of the users and apply revisions that are more durable and efficient. This part of an evolving workflow management system is called *learning by supervision*. Together, these learning cycles form a *double-loop learning approach* that is depicted in Figure 3.4.

[6]The deficiencies of a regular workflow management system were presented in Section 1.1

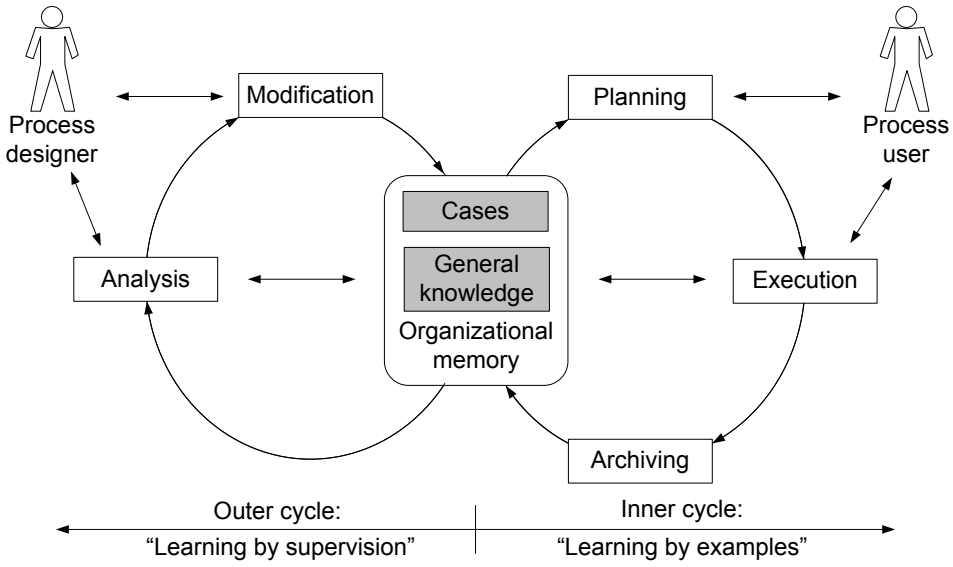


Figure 3.4.: Double-loop learning (compare [WWT98] Figure 1)

Central component of double-loop learning is the organizational memory. One part is a case base filled with terminated workflows and related audit data. The other part bears domain specific knowledge. Both learning loops are attached to the organizational memory, "learning by supervision" as the outer cycle and "learning by examples" as the inner cycle. The process designer analyzes cases within the organizational memory by observing their structure and frequency and modifies process models accordingly. Process users plan their activities and process executed instances. For treatment of upcoming exceptions during execution, they rely on the organizational memory. Subsequent solutions for exceptions are archived as cases in the case base.

In order to simplify usage, partial processes called "single building blocks" are stored as cases rather that entire processes. By this modularization, the number of variations within cases decreases significantly. Building blocks exist on various level of granularity, that is, blocks are based on a varying level of detail. The lowest level represents an elementary activity, intermediate levels are sets of activities or subprocesses. The top level is represented by a complete process phase. For each level, a catalog describes the available cases within the case base.

A further implemented concept is separation of control: Organized in two layers, a management layer defines milestones, beneath actions are self-managed by subunits and users in a second layer. On the management layer the process is automated and for subunits an ad-hoc changeable subprocess allows optimization and more control on details.

The advantage of using an OMIS is that it integrates processes and information. Knowledge becomes explicitly stored and does not get lost when employees leave an organization. Both parties, process users and designers apply changes to process instances and types respectively. Thus processes can adapt to changing internal and external require-

ments. Due to the outer learning cycle, continuous analysis makes it easy to check for performance, goal attainment and to apply benchmarking and monitoring methods.

The WorkBrain approach appears very similar to CCBR: Both improve the process incrementally by allowing the user to make adaptions according to his current situation using a case base. However, in contrast to CBRFlow, WorkBrain interprets and uses the case base in a broader sense as a part of an organizational memory. While in CCBR a case base is a set of deviations, exceptions and adaptions, in WorkBrain it contains both fundamental and supplemental data.

Exception handling with CBR Hwang et al. aim in [HHT99] at supporting users on exception handling with a case base of past exceptions. When a new exception shows up, the system proposes solutions of previously experienced exceptions based on similarity measures.

Two types of exceptions are differentiated: Those that are *expected* and are treated by implementing explicit modeling for exceptions. These are typically adaptive workflow approaches, such as ADEPT [RD98]. The other type of exceptions which occur completely *unexpected* are dealt with in this approach: Reactions for exceptions include ignoring them, retry the failed activity, perform a partial rollback of the process, add compensation activities or delete planned activities and make general evaluations for correctness.

Their idea of exception handling has two steps: First, a rule base is consulted. Each rule refers to a type of exceptions. If a matching rule could be found, then the rule tells how to resolve the exception. The rule base represents domain specific knowledge that makes it possible to specify instantly the correct treatment of a predefined type of exception.

If the rule base fails, then in a second step the user searches through a case base filled with exceptions for similar exceptions and their resolution. Search is done by a similarity metric, that is based on a number of attributes that characterize a particular exception:

- **Process instance status** holds the execution state of a process instance and all its constituent activities (compare Section 2.4.2)
- **Activity** describes which activity caused the exception
- **Event** gives a semantic description of the type of exception specified by keywords and/or free text
- **Who** experienced/noticed the exception?
- **When** in time did the exception happen?

A complete matching between all attributes of the current exception and an stored exception in the case base is very unlikely. In order to implement a similarity measure,

a key idea of this project is to formalize and classify the relevance of attributes and their influence on the similarity of exceptions in *concept hierarchies*. For each attribute type, concepts are ordered hierarchically in a tree structure such that the most general concept is the root of the tree and more specific concepts are organized in branches. Within hierarchic trees, the *depth* of a node denotes its distance from the root. *Levels* in contrast describe the depth of a node i with regard to the tree's lowest leaves. That is, $level_i = depth_m - depth_i$ with $m = max\{depth_n | n \in T, n \text{ is a node}, T \text{ is a concept tree}\}$.

Example 19. A concept hierarchy for the attribute *Activity* corresponds to a task decomposition of the activity as shown in Figure 3.5.

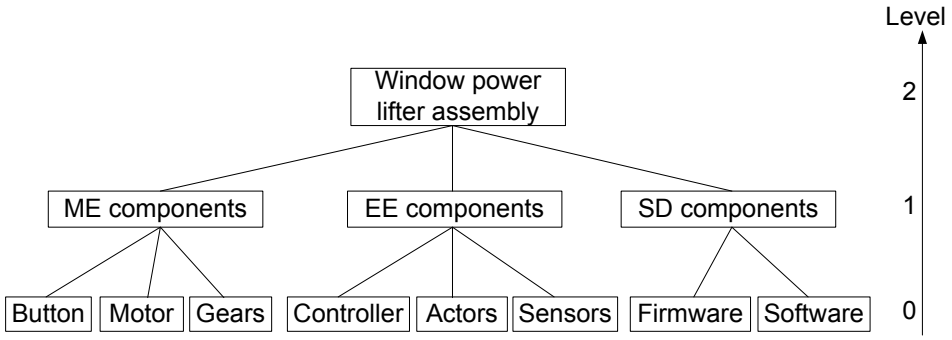


Figure 3.5.: Example: Concept hierarchy of a window power lifter assembly

The assembly of a window power lifter constitutes components from the three disciplines mechanical engineering, electrical engineering and software development. The leafs of the tree represent the lowest level 0. With each parent node, the level increments.

Let the tuple $< a_1, a_2, \ldots, a_k >$ be a number of k attributes that describe the current exception. This is matched against the attributes of solutions $< s_1, s_2, \ldots, s_k >$ in the case base. A function $leastCommonAncestor(< a_i, s_i >)$ takes two attributes as input and returns the least common ancestor l of both in the concept tree. That is, it returns the node l in the concept tree at the lowest level possible, that contains both a_i and s_i in its child branches. The level of l is an measure for the similarity of the compared pair of problems and solutions with respect to the i^{th} attribute: The lower the level of l, the more similar is the solution to the given problem.

Example 20. This example illustrates the *least common ancestor* concept building on Example 19. Suppose a problem is described among others by an attribute a_i that specifies the activity type. Two solutions s and s' are found in the case base. The values for each one of these attributes are shown in Figure 3.6.

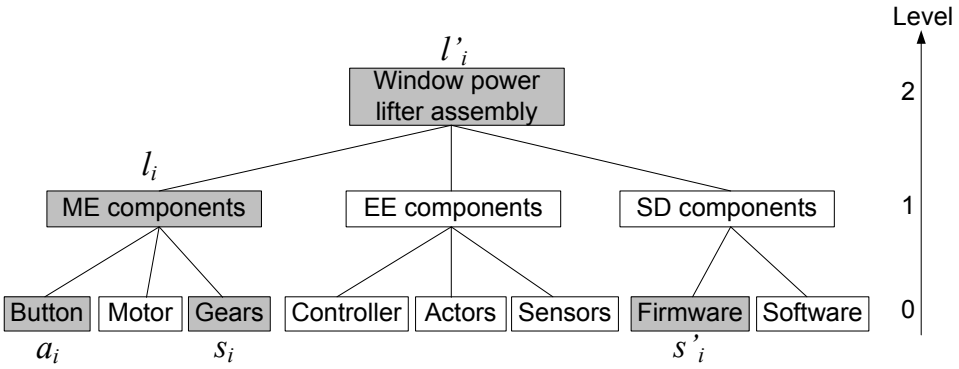


Figure 3.6.: Example: Least common ancestor in a concept hierarchy

The least common ancestor l_i for $< a_i, s_i >$ and $< a_i, s_i' >$ has level 1 and level 2 respectively. Because of its lower common ancestor level, solution s_i is preferable over s_i': This makes sense because an exception that occurred during the assembly of gears is more similar to an exception on button assembly than an exception during firmware upload.

If additionally a weight function is defined, the relevance of each attribute can be weighted individually. Adding the weighted attributes up returns a similarity index between the current exception and the solution. Based on this index, the best matching solution is proposed. The best solution with the lowest overall index matches as the most relevant solution on the lowest level in the concept tree.

Apart from efficiency and implementational issues[7], the concept tree approach is limited by two issues: First, a weight function influences the matching process heavily and has to be set for each upcoming problem individually in order to optimize results. Finding a generic weight function that matches most problems well might to turn out to be very hard. Second, the foundational assumption of concept trees is that all attributes can be broken down hierarchically. This is not necessarily true for all types of attributes and can be ambiguous, too. An *Activity* attribute might be decomposed either in a more logical fashion or a way that reflects better the temporal aspects of the tasks that constitute the activity. The first would be formally more correct, while the second reflects better on reality.

[7]Implementational issues are discussed in [HHT99] in detail

3.1.3. Assessment of usefulness

CBR in general bears many useful aspects for Emergent Workflow: Its learning & reuse-based approach has benefits for handling recurring problems more efficiently. In application, it has been identified to be well suitable for slight variations around clearly identifiable tasks. As an integrated approach, CBR must be combined with new case creation methods in order to be useful.

CODAW on page 70 introduces multiple interesting aspects for Emergent Workflow: Its XML based process model and instance represents a lightweight alternative to BPEL, XPDL and other standardization efforts. In fact, this may be due to the fact that CODAW including its case representation has been developed for an engineering application field which allows to reduce generic constructs. With respect to the Emergent Workflow requirements summarized in Table 2.4, Section 2.3.10, these representations can be used for temporary instance representation (R1a, R1b) and permanent storage of process models (R2c) and fragments (R2d). In particular, this instance representation allows annotations (RE9) (this feature is here meant for administrators, but could be used by users as well). Its state-space declarative representation allows the representation of task compositions (R2e). As the representation is XML based, basic access methods like XQUERY for full-text search exist(R4). Furthermore, the similarity flooding schema matching algorithm is versatile enough to be used in the process matching engine for searches with qualitative constraints (PM1). It performs an approximate, schema-matching search (PM2, PM3) on syntactic similarity (PM4) and allows filtering of its outputs (PM5). The planning algorithm SHOP can be used for automated case selection and planning supporting the recognition of cases (RE5). As it is able to compose and decompose composite tasks, it may be used by process designers for fragment composition (UI1).

CCBR on page 75 improves human-machine interaction: Fundamental to this approach is the separation of automatic rules and manual cases. A mixed-initiative, conversation-based case finding enhances usability of a system (UI2). Furthermore, reinstantiation of a defined case is done manually but already predefined (RE5). It invokes an exact matching algorithm (PM2) based on descriptive and quantitative attributes and constraints (PM2, PM3). The analysis of the case base is done manually (RE5).

WorkBrain on page 77 introduces the concept of an Organizational Memory Information System. This integrates general knowledge with cases representing archived instances (R2d). The dictionary and organizational model in Emergent Workflow can be seen as a reduced version or a part of the organizational memory (R2a, R2b). Double-loop learning addresses the evolution of process models initiated by workflow users (RE6, RE9) and revised and permanently implemented by process designers (RE4, RE7).

The work by Hwang et al. on exception handling on page 79 introduces exception handling (RE6) by using a rule base for automated handling and a case base that supports manual resolution of exceptions (RE5). Based on their idea of concept hierarchies, this

approach performs similarity matching between exceptions using exception attributes. This is a quantitative (PM1) and exact (PM2) matching type based on a descriptive (PM3) search for structural similarity (PM4). The concept of least common ancestors allows to put an index on results and to sort them (PM5).

3.2. Process mining

3.2.1. Fundamentals

Explicit creation of process models is a lengthy task and its outcomes do not always reflect the real process accurately. The goal of *process mining* is to reverse the process and collect data at run time to support workflow design and analysis. Van der Aalst et al. describe *process mining* respectively *workflow mining* in [ADH+03] as follows:

> *The term **process mining** refers to methods for distilling a structured process description from a set of real executions. Because these methods focus on so-called case-driven process that are supported by contemporary workflow management systems, we also use the term **workflow mining**.*

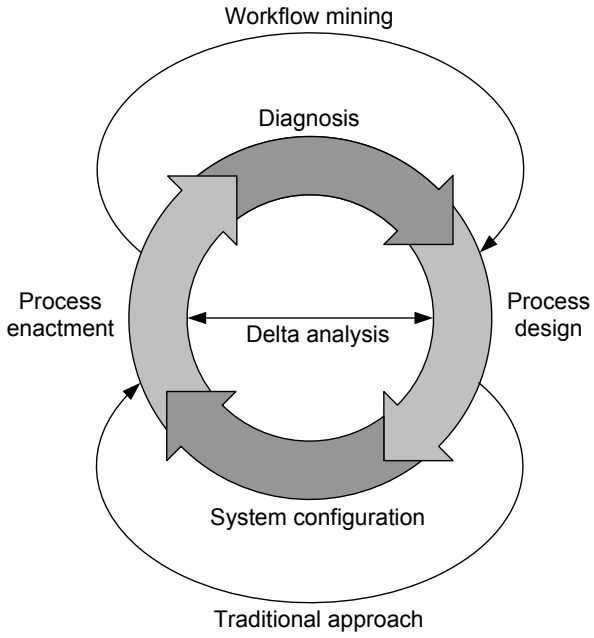


Figure 3.7.: Workflow mining in the business process life cycle (compare [ADH+03] Figure 1)

The business process life cycle has already been introduced in Section 1.1 (Figure 1.1 on page 2). In Figure 3.7, the role of workflow mining is shown in the context of the business

process life cycle. While the traditional approach starts with process design and develops models which are later enacted, workflow mining takes the outputs of process enactment and supports process design. This is possible because implicitly processes do always exist, even if they were not explicitly modeled in a preceding design phase. Involved software such as an ERP system usually keeps track of events and transactions and provides logging functionality. Process mining can use this information as a starting point for the derivation of a formalization of the ongoing process. A *delta analysis* compares bidirectional the designed process models with mined real processes from the enactment phase. This "delta" gap between a model and its actual behavior shows discrepancies that can indicate weaknesses of models. These are very useful e.g. for iterative process improvement.

Input & output A minimum **input** for a workflow mining algorithm is a sequential list of entries describing an event of the execution of a process instance each. This list is also referred to as a *log*. A log file must meet following minimum requirements:

- Each log event refers to a task
- Each log event refers to an instance
- The log events are totally ordered

A minimal log is shown in Table 3.1.

Instance identifier	Task identifier
Instance 1	Task A
Instance 2	Task A
Instance 1	Task B
Instance 2	Task C
⋮	⋮

Table 3.1.: Simplistic minimum activity log

When real-life information systems record protocols of events, usually much more information is put into a log file: For each event, an *event type* is specified such as "start/stop". Additionally, a time stamp as well as further context-specific can be supplied for each entry. Such would be a transaction status, indication of exceptions, a user causing the event or a case description within a certain instance. Notice that such additional information is necessary for a more sophisticated semantic analysis as Emergent Workflow intends to do (see Section 2.3.6).

The **output** of a mining algorithm is a representation of a process model or an incomplete template indicating its schema. The outcome determines how much of the business process cycle can be skipped by mining (see Figure 3.7 on page 83) a log file. Workflow

mining is either able to step over the whole process design phase if resulting models are ready for enactment or gives a starting point for process designers who revise a provided template.

Problem definition Mining a process graph can be seen as two subproblems:

- **Finding of a schematic graph structure that generates the log output.** The extraction of a set of structural dependencies (which are usually visualized in a process graph) from a set of logged events is what most mining algorithms are dealing with. With some limitations, this can be done based on a minimum input log by a syntax-analyzing algorithm without user interaction. Notice that most algorithms do not focus on the reconstruction of the *exact* generating graph but create a *sound* and *equivalent* graph which has the same output. That is, any reachable marking state in the graph is legal and terminates correctly with the same functional result as the original.
- **Finding of edge conditions.** The second step of recreating a process model is less straightforward as it requires an understanding of the semantics of a process. Real processes make use of conditional constructs such as exclusive branching and loops. During the first step, the fact that a conditional construct exists has been detected – now it has to be added *what* the condition was. If this is planned to be done by information provided by the process log only, then the process log must be enriched with supplemental data describing a task's and instance's context. For example an exclusive conditional splits up two distinct alternatives identified by a condition. If these two cases can be identified, then it is possible to derive a condition.

Complexity & Incompleteness One can look at a graph that represents a process model PM as a finite state automaton[8]. The process schema defines here a grammar on an alphabet $\sum$ whose symbols are tasks. A formal language L on $\sum$ is then defined by any subset of $\sum^*$. L consists of all words that can be generated by a given grammar over $\sum$. Then there exists an onto[9] function that maps each process instance that was enacted on PM to a word of L. The existence of such a function is evident because instances are not only specified by their sequence of tasks, but also by their context. What process mining actually tries to do is to draw a conclusion on the process schema from a limited set of instances. As real-life business processes are rather large, this set is almost certain to be incomplete. This situation equals to having an incomplete set of words of an unknown language. Now one tries to guess a grammar that creates the unknown language. This procedure is very unlikely to yield a correct guess. Therefore,

[8]See any book introducing language theory, e.g. M. Sipser "Introduction to the Theory of Computation".

[9]Note to German readers: "Onto" translates into German "surjektiv".

there will be always a difference between the real process and its mined reconstruction, even though mining techniques attempt to make a very realistic guess that matches the original process model for certain classes of processes quite well.

Example 21. In this example, the conceptual idea of the α-algorithm [AWM03] is given: The α-algorithm inputs an event log as shown in Table 3.1 and outputs a Place/Transition net (P/T-net). P/T-nets extend the Petri Net formalism for use with multiple concurrently running tokens[a].
Fundamental for most mining algorithms is the idea of *causal relations*. It is defined as follows: An activity B *follows* an activity A ($A \rightarrow B$) if either B starts after the termination of A or there exists an activity C such that C follows A and B follows C ($A \rightarrow C \wedge C \rightarrow B$) in each instance log. If $A \rightarrow B \wedge B \nrightarrow A$, then B *causally follows* A. If $A \rightarrow B \wedge B \rightarrow A$ or $A \nrightarrow B \wedge B \nrightarrow A$, then A and B are *independent*.
The basic functionality of the α-algorithm is the following: A task *exists* in the resulting net if it appears in any log trace. A task is either the first task of a process model or has an ingoing edge for each task that this task *causally follows*. In an analogy, a task is either the last task of a process model or has an outgoing edge for each task that *causally follows* it. If a task is neither the first or last task in a process model nor does it have any causal relations, then it does not receive any ingoing and outgoing edges. This version of the α-algorithm mines simply structured graphs (including sequence, parallel and conditional branching) mostly correct, but fails on structures containing short loops, invisible or duplicate tasks and other advanced constructs. Support for them requires extensions which are further discussed in [ADH+03, MAW03].

[a] See W. Reisig and G. Rozenberg in "Lectures on Petri Nets I: Basic Models", volume 1491 of "Lecture Notes in Computer Science".

Difficulties Besides the conceptual problems of process model recovery from log files, additional conditions complicate the functionality of process mining.

Noise in process logs describes the fact that process logs can be not only incomplete, but also incorrect. Due to human or technical errors, a log file is possibly disordered or events themselves contain wrong information. Even with correct input logs, wrong models can be mined due to coincidentally colliding events that are not related but are misinterpreted by mining algorithms. Certain mining approaches try overcome this by the introduction of stochastic models and frequency tables that help to detect and ignore erroneous entries[10].

Privacy is a non-functional issues that has major impact of the usability of process mining. As an event log contains personalized information about individuals interacting with an information system, the storage and processing of process logs may be subject to restriction due to federal legislation or corporate ethics. Functionality to anonymize data before collecting it in an event log may be required in certain situations.

[10] Compare the work of Herbst and Karagiannis referred to in [ADH+03]

3.2.2. Multi-phase process mining

Van Dongen and van der Aalst present in [DA04] a process mining approach in a control-flow perspective, that creates visualizations of individual process instances. It splits the mining process up into two phases: The first step creates representations for each running instance individually, the second step optionally merges the instance representations into an overall process model.

This is motivated by the fact that, during run time, analysis of performance is interesting, such as the average time to transfer a task from one person to another. The implemented processes however differ from actual execution, therefore their analysis is not sufficient. Rather, individual execution trails are discovered by mining an individual process instance history from process logs. So the basic idea is to look at each instance individually rather that looking at a combined, overall trace of events.

Without giving formal specifics, *instance graphs* are created as follows: A process log consists of a sequence of log entries that refer to multiple process instances. Processes are mined by first extracting an *instance net* and transforming that into an *instance graph*. An instance net is based on an *instance domain* which links each log entry to a task. This is necessary as duplicate tasks may appear in a log file. The instance domain indexes the log entries and enables their clear referral. The instance net is an ordered set of log entries which stem from one process instance. As an instance net has already been executed in the past, no choice or loop constructs are needed. The properties of this ordering relation (referred to as $\diamond$ in the following list) are:

- $\diamond$ is irreflexive, asymmetric and acyclic
- If an entry i appears before an entry j in the log, then $j \diamond i$ can not exist
- For any $i \diamond j$ there is no common intermediate element k such that $i \diamond k$ and $k \diamond^+ j$. The symbol $\diamond^+$ expresses that there may exist any sequence of $0 \ldots n$ intermediate ordering relations $\diamond$ between k and j.
- If duplicate tasks appear in the log, then they must be related with $\diamond^+$

This ordering creates relations between the closest tasks, each of which have a causal relation. As a causal relation indicates a sequential structure, no symmetric causal relations with the exception of short loops are allowed.

Creating an instance graph from an instance net is straightforward: Each task from the instance net represents a node in the instance graph and each causal relation creates an edge. If tasks have no causal relation with their predecessor or successor, their node representation are parallel branches. Due to the retrospective view on the process as a log, choice branching is not supported. Finally, a start and end node are added with in- and outgoing edges respectively going into nodes that have no predecessor or successor. An instance graph holds the property of being strongly connected. Furthermore, an

entry in the log only appears if all its predecessors in the directed graph have already appeared in the graph[11]. These assure the correctness of the reconstructed flow of tasks in terms of executability and conformance with the records provided by the process log.

Instance mining may be beneficial when process logs are not complete as their completeness is not required to produce useful results. Primary ways of usage include instant instance visualization and other related functionality supporting the analysis, control and planning of processes. As a secondary option, an instance graph can be either used to be transformed into other data formats[12] which may offer further processing such as the aggregation of multiple instance graphs into one process schema. Usage limitations apply when dealing with erroneous logs that require preceding filtering steps. Moreover, meaningful aggregation is hard to accomplish when more complex routing structures are involved.

3.2.3. Assessment of usefulness

The process mining approach clearly aligns with the functional requirements of the process creation engine of Emergent Workflow (see Section 2.3.6). All requirements references in paranthesis refer to the requirements summary in Table 2.4, Section 2.3.10 if not noted otherwise.

In Emergent Workflow, the audit trail composed of user interaction and system events represents an event log enriched with context data (see Section 2.3.7 on page 33) which meets the process creation engine's input requirements (PC1, PC I/O). Regular workflow mining as described on page 83 proposes algorithms that require as input completed logs of sufficiently many instances in order to function properly. Therefore, they are focused on ad posteriori analysis. With respect to the different purposes of documentation shown in Table 2.2 on page 29, workflow mining offers means to document for later reuse, but does not support planning or synchronization of ongoing operations (PC2). In order to achieve robustness against real-life circumstance such as noisy log files, advanced mining methods have been proposed (PC3).

Multi-phase process mining on page 87 is a mining approach of particular interest for Emergent Workflow. Its first phase performs individual instance mining which aligns perfectly with the creation of process fragments by a process creation engine (PC2). It is not as complicated as regular process mining because it restricts itself to log analysis of single instances. Thereby, the outputted metamodel is simpler as for example no conditional branching is allowed and necessary (PC4). Moreover, its output format can be easily transformed into other representations (PC4). Most noticeable is the fact that instance-based mining is useful for immediate and individual support for workflow users. While regular workflow mining is rather a post-hoc analysis, instance-based mining can be done during execution as it does not require completeness of its input (PC2, PC3).

[11] Compare [DA04] page 8/369 for proofs
[12] Such as an Instance Event-driven Process Chain in [DA04] page 8/369 et sqq.

Finally, one can compare the results of instance-based process mining with the planned overall process. This allows an analysis of the flexibility of process models and their average level of deviation from the planned process.

3.3. Flexibility approaches

Approaches that introduce flexibility on process models during run time can be classified into two categories: *Ad-hoc change of process instances* applied to instances during run time and *schematic changes* applied to process models. As already motivated in Section 2.3.7 on page 40, instance-based changes are used for exceptional situations or changes affecting only selected instances. Schematic changes indicate incremental systematic change that applies to all instances and causes the process type to evolve.

3.3.1. Schema evolution and propagation

Schema evolution consists of a *static* part modifying the process models and a *dynamic* part which refers to managing the migration of running instances [CCPP96].

A *static* evolution is the issue of modifying the workflow description and includes checking for syntactic correctness. *Dynamic* evolution refers to managing running instances whose type was modified. They require some form of assistance to adapt to the new requirements formulated by the type change. Their consistency regarding their execution state needs to be checked and assured.

Change operations on a process can have an impact on any one of its perspectives (see Figure 2.14 on page 54): For instance the assignment of tasks to users or the organizational structure can change as well as associated applications and use data. The modification of the *control flow* is focused in the following Section due to its high relevance.

A set of operations modifying the control flow holds characteristics such as being *complete, minimal* and *consistent*. Completeness is achieved if any schema can be transformed into any other schema. Minimality refers to the fact that only a minimum set of operations is offered that meets the completeness requirement. Consistency means that the change operation reinduces no errors during run time.

Dynamic schematic changes occur during workflow execution when the process model adapts to a changing environment. Possible strategies to handle these changes during execution are:

- **Flushing the system.** The enactment of new instances is delayed until all running instances have terminated. Then changes are applied and enactment is

restarted. This strategy is safe but time costly and not acceptable when dealing with many and long-running instances.

- **Abortion of all jobs in progress.** Running instances are aborted, the process model is altered and instances are re-run using the new schema. Again, this strategy is unacceptable due to the high costs of restarting all instances and redoing all the work to reach the originating state.

- **Run old and new versions simultaneously.** Here, running instances remain running on the "old" process model while newly enacted instances use the new process model. The old process model remains active until all old instances have terminated. This strategy is potentially unsafe and inconsistencies are especially likely if the schematic change interferes with data dependencies or the change downsizes the model (see below).

- **Safe migration of instances from one version to another.** The change is applied to the process model and running instances are individually migrated to conform with the new process schema. Here, safety is an issue because correctness and consistency need to be checked explicitly.

Obviously, a safe migration of executed instances upon schematic change is the most challenging and promising strategy at the same time.

Synthetic cut-over change Ellis et al. deal in [EKR95] with the dynamic change problem and ways to verify the correctness of one class of dynamic change. They present a certain class of processes for which the consistency of migrated instances can be proved. Their approach is to define a *change region* as that part of a process model which is being affected by a structural change. The *old change region* existing prior to the change is then replaced with a *new change region* containing the change while obeying the procedural specifications in order to maintain correctness. Correctness is maintained if all instances resume and finish according to either the old version or the new version of the procedure. A special class of changes referred to as *synthetic cut-over change* is observed when the new change region contains both the old and the new region.

A Petri Net formalism[13] is chosen to represent process models as marked networks. In Petri Nets, a change is a replacement of a marked subnet by another marked subnet. The *old change region* is defined as the smallest net containing all activities affected by the change operation. Those parts of the net connecting the change region to its context are described as the *interface*. Thus communication between change-affected and non-affected regions is restricted to the interface. Intuitively, the changed network is obtained by removing the old change region from the network and plugging the new change region into the interface.

[13] An basic introduction to the Petri Net formalism is given e.g. in [EKR95] p. 14 et sqq.

Dynamic change correctness with respect to the used formal model splits in three key issues: *Fault prevention* means to disallow any changes such that the marked network can not reach a terminal (final) marking state. Assuming that the initial marking of the old and new network both comply with the fault prevention property, a *system replacement* which cancels all instances in progress and restarts everything should maintain correctness as well. If the system is not restarted, then the *consistency* of hybrid executing sequences needs to be assured. A hybrid sequence consists of a pre-change sequence and a post-change sequence which is supposed to continue the work initiated before the change. Hence, each marking state that leads to a valid terminal state in the old network must do so as a pre-change part of a hybrid sequence on the new network as well. Additionally, all hybrid sequences must be valid execution sequences of the new network.

A dynamic change can be either *immediate* of *delayed*. In the prior case, any change operation takes effect on all involved instances immediately as the change region is replaced and existing tokens representing instances have immediately migrated into a new schematic environment. The proposed solution to delay a change operation is motivated by increased safety in certain cases. The idea named *synthetic cut-over change* is to maintain the old and new change region both at the same time within the process model. Already existing tokens in the old change region practically do not take notice of the change operation whereas new tokens entering the change region will only get in touch with the new change region. The change appears to be immediate for all tokens but those in the old change region. The following example visualizes a synthetic cut-over change.

Example 22. Suppose a product development process. Part of this process is the construction of a component. As depicted on top of Figure 3.8 and named "The old change region", the activity "Construction" is followed by component integration and simultaneously the analysis of upcoming problems. Upon the completion of both activities, an interdisciplinary meeting is held in order to discuss the encountered problems. Notice that the shaded circles indicate the interface of the change region. Let us now assume that in this scenario the parallel processed activities "Integration" and "Problem analysis" are changed into a sequential order "Integration", "Problem analysis". In order to achieve a delayed change, the old change region is transformed in the new change region shown on the bottom of Figure 3.8. It consists of the old change region *and* the new sequential procedure whose output interface is connected. This assures that newly generated tokens traverse the new schema whereas existing tokens in the old change region will not notice the change.

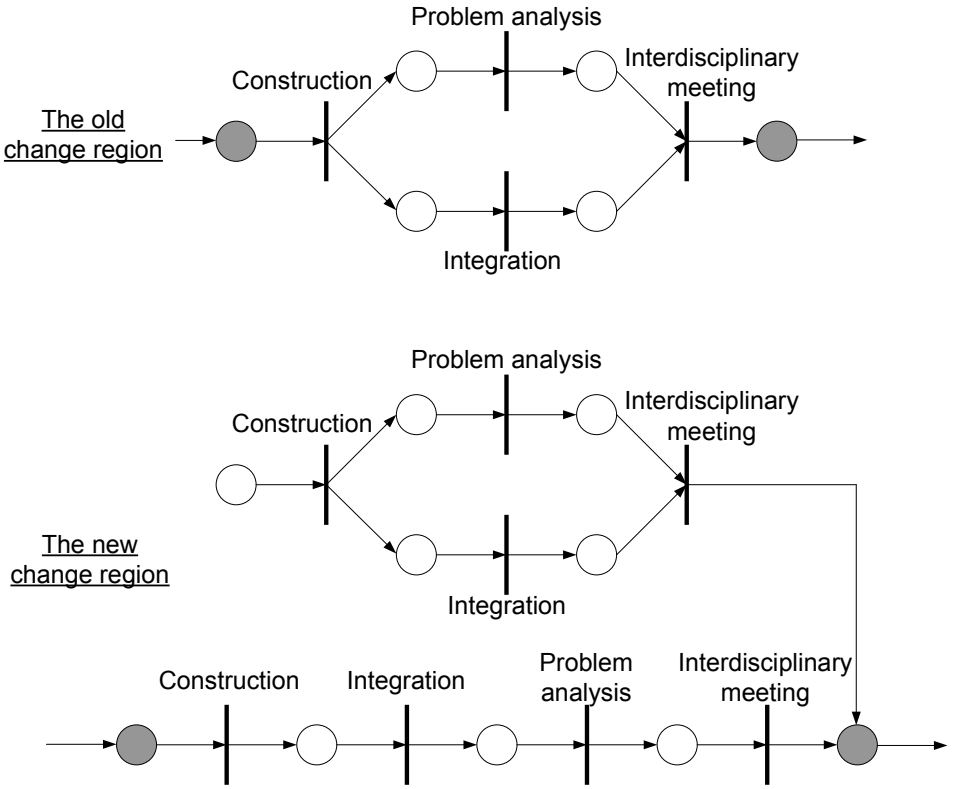


Figure 3.8.: The old and new change region in the case of a synthetic cut-over change

The formal distinction between immediate and delayed changes is justified by differing change safety. Change regions can be split up into a number of elementary operations. Depending on the change operation, properties called *upsizing* and *downsizing* can be informally established as follows: If the new change region contains all elements of the old change region (it can "do more" such as the insertion of a new activity), then the change is called *upsizing*. In the reverse case the old change region contains all elements of the new change region (e.g. a delete operation) and the change is *downsizing*. Ellis et al. prove[14] the correctness of any immediate upsizing change. However, only the delayed version of a downsizing change is always provable correct.

3.3.2. Ad-hoc instance change

Ad-hoc adaptive workflow with ADEPT In static workflow management system, process designers create process models and take responsibility for producing only models whose instantiations run and terminate correctly. When alterations are made to them spontaneously by users, correctness and consistence is usually no longer guaranteed.

[14] Compare Ellis C., Keddara, K., Rozenberg, G., "The Modeling of Dynamic Change Within Workflow Systems"

ADEPT[15] presents a framework that allows a user to perform ad-hoc changes on running instances without shifting the responsibility for correctness to him. ADEPT$_{flex}$ by Reichert and Dadam [RD98, Rei00] contains a set of change operations applied to process instances and foots on the designated ADEPT workflow model.

The ADEPT workflow model holds a number of characteristics which are essential for the functionality of dynamic structural change methods. Fundamental to the structural design of ADEPT is its concept of *symmetrical control structures*. It means that tasks are partitioned into symmetrical blocks with well-defined start and end nodes. These blocks are not allowed to overlap but can be nested. Elements of control structure are applied to whole blocks in the same way as they are applied to primitive tasks. In the following paragraphs, an overview of ADEPT's control flow, data flow, change management and undo capabilities of temporary changes is given.

ADEPT's **control flow** is represented by a directed structured graph. Available basic control structures are *sequence* and *parallel processing* (n-of-n split and join), exclusive *conditional routing* (1-of-n split and join) and *parallel branching with final selection* (n-of-n split and 1-of-n join). It does also provide advanced control structures such as *loops*, *failure edges* and *synchronization edges*. *Loops* allow cyclic structures within the process graph by inserting a loop edge that connects a unique start node with a unique end node within a block. A loop condition at the end node is used to check whether the loop edge or the next task is chosen next. A *failure edge* is a second outgoing edge from an activity n_{failed} that points to another activity n_{restart} that precedes n_{failed}. This edge is signaled on failure of the activity and resets all activities succeeding n_{restart} and preceding n_{failure}. *Synchronization edges* are introduced in order to enable synchronization of tasks from different branches that are processed in parallel.

A control flow is considered *correct*, if from every reachable state a correct terminal state can be reached by a number of valid state transitions (*safeness*) and each node is reachable by a number of valid state transitions from the start node (*reachability*).

Data flow in ADEPT constitutes *data elements*, *I/O parameters* and *auxiliary services*. *Data elements* are global elements within a workflow representing data objects that are collaboratively read and wrote by tasks. *Input and output parameters* of tasks referring to data elements define the data flow within a workflow schema. As various tasks implement different data input and output formats, *auxiliary services* are meant to provide a common interface to data elements for all tasks. They are individually associated with each task and transform data inputs and outputs accordingly.

In order to uphold correctness with regard to the data flow, all input and output parameters and auxiliary services have to be available in time. That is, input and auxiliary services are required to be ready before execution and output before termination. Globally accessible data elements bear the possibility of data inconsistency if tasks manipulate data elements concurrently without synchronization. Therefore, tasks of an instance

[15] ADEPT stands for Application Development Based on Encapsulated Premodeled Process Templates

work on individual copies of the data element instead of the original. Upon successful task termination, the global data element is replaced with the most recent version but not discarded though. This holds two advantages: First, tasks in parallel branches can work independently on local data copies. In order to maintain correctness, their updates on termination must be synchronized. Second, in case of a rollback (which is an essential exceptional scenario within flexible workflow management as further discussed below) data elements can be reset to their initial state as their history is still available.

ADEPT$_{flex}$ represents a set of operations that allows dynamic schema changes on running workflow instances. Analogous with properties presented for schema change operations, the main focus designing these operations is put on the following three properties:

- **Correctness/consistency**: The application of a change operation to a workflow instance should neither affects its structural schematic correctness nor the consistency of its execution state.
- **Adequacy/completeness**: Each change operation should be applicable to any kind of correct and consistent workflow instance. Completeness is met if any kind of structural change can be achieved by the application of a sequence of basic change operations.
- **Minimality**: The set of operations is minimum if the removal of any operation violates the completeness requirement.

ADEPT$_{flex}$ consists of the following basic operations:

- **Insertion** of a task into the process graph
- **Deletion** of a task from the process graph
- **Changing task sequence** during run time

These are used to skip tasks for fast forwarding, to jump to currently inactive parts of the process graph, to serialize previously parallel tasks and to rollback and undo temporary changes. Higher level operations can be achieved by repetition and/or a composition of these basic operations. For example an ad-hoc workflow definition can be achieved by starting with an empty workflow and applying an insert operation repetitively on it.

Change management Problematic scenarios can arise when multiple workflow instances are changed concurrently. Exemplarily a few are mentioned: For instance different changes can be made to multiple instances of the same type concurrently. Changes can also be made to an already changed type. Some changes may require secondary additional changes (*concomitant changes*) in order to preserve correctness and consistency of the underlying workflow model. Finally, there exist changes that last only temporarily and have to be undone some time after their application.

In order to enable proper handling of such scenarios, each workflow instance p_i maintains the following information:

1. A process graph P_{all} representing the current process schema which includes all changes and state information of p_i.
2. A process graph P_{perm} whose graph structure contains only permanent changes – temporary changes as well as state information is left out.
3. A *change history* C which is a chronologically ordered vector of all *changes* applied to p_i. Each change record consists of the following information:
   - The *type* of a change operation
   - The *durability* of a change (can be either temporary or permanent)
   - The *initiator* of a change
   - The *start region* of the change in order to determine whether and when to undo a change
   - Additional *concomitant changes* to maintain correctness/consistency
   - A list of the applied *change primitives* in order to break down change operations into graph modification primitives

Temporary changes c_t are done by first checking for correctness and consistency after their application to P_{all}. If unresolvable problems persist, the user has to resolve exceptions an other issues manually. The change is performed and it is added to end of the change list C. Permanent changes require consistency checking for P_{all} as well as P_{perm} before a change operation c_p can be applied to both of them.

Undo temporary changes Particular changes can be undone by removing them from the graph structure P_{all}. Part of each change record is the definition of a *start region* that describes a set of nodes in the process graph: If each node within the start region is within a terminal state, the undo function of the temporary change is triggered.

Undoing a temporary insert or delete change operation c_l works similar to the roll-back/recovery concept of a transaction oriented system[16] as visualized in Figure 3.9: A change list consists of n sequential changes where c_1 represents the oldest and c_n the latest change. As undoing c_l can cause a state change for a set of nodes (the so-called backward region), other changes whose start region overlaps with the backward region need to be undone as well. (1) Hence, the oldest change c_k ($1 \leq k \leq l \leq n$) whose start region interferes with the backward region of c_l has to be found. (2) Then both permanent and temporary changes are undone in reversed order starting from c_n up to c_k. (3) Finally, all permanent changes between c_k and c_n are redone in forward direction.

[16] Compare e.g. the lecture notes on "Database Systems – winter term 2003", University of Ulm

Temporary changes are redone if their start region is not covered by c_l's backward region and correctness and consistency of P_{all} remains.

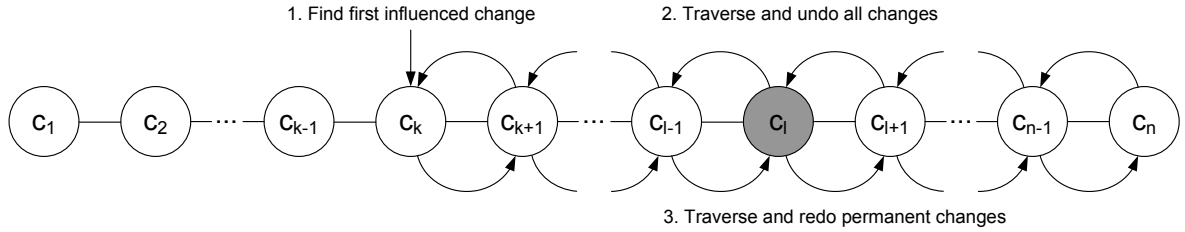


Figure 3.9.: Undoing a change within a change list

3.3.3. Integration of schema evolution and ad-hoc instance modification

Process-aware information system (PAIS) A very recent approach by Weber, Rinderle et al. [WRWR05, RWRW05] named PAIS proposes the integration of the systems ADEPT (see Section 3.3.2 on page 92) and CBRFlow (see Section 3.1.2 on page 75) which were introduced earlier in this thesis. Its goals are to offer reusability of instance-based ad-hoc changes and to accomplish a derivation of evolutionary schematic changes.

ADEPT contributes to this composite project with the abilities of a full-feature workflow management system including modeling, analysis, execution and monitoring capabilities [RD98]. As already mentioned before, ad-hoc change functions are provided by ADEPT. Additionally, it offers schema change operations for process types. Its process representation based on symmetrical control structures allows on-the-fly migration of running instances while preserving process consistency for most classes of instances (see Rinderle et al. [RRD02, RRD04]). The shortcoming of this system is however that its ad-hoc adaptions are not reusable.

CBRFlow contributes a case-based reasoning (CBR) approach including all of CBR's characteristic features (see Section 3.1). It documents the reason for instance changes and makes them reusable for the future.

This joint approach now aims at covering the whole *process life cycle* with a combination of both functionalities: Figure 3.10 illustrates how ad-hoc changes and type changes are integrated with reuse of altered instances in a case base. From a given schema, process instances are created. As now the user experiences an exception during run time, he requests similar cases from the case base and either retrieves a matching case or adds a new one. Deviations are modeled with change constructs and a documentation on the case is added which makes the case immediately reusable. The reuse of existing cases is counted and in case a defined maximum number (the threshold) is exceeded, process designers are triggered with a notification indicating the possible need for a process type change. In case the process type was updated, existing cases in the case base must be

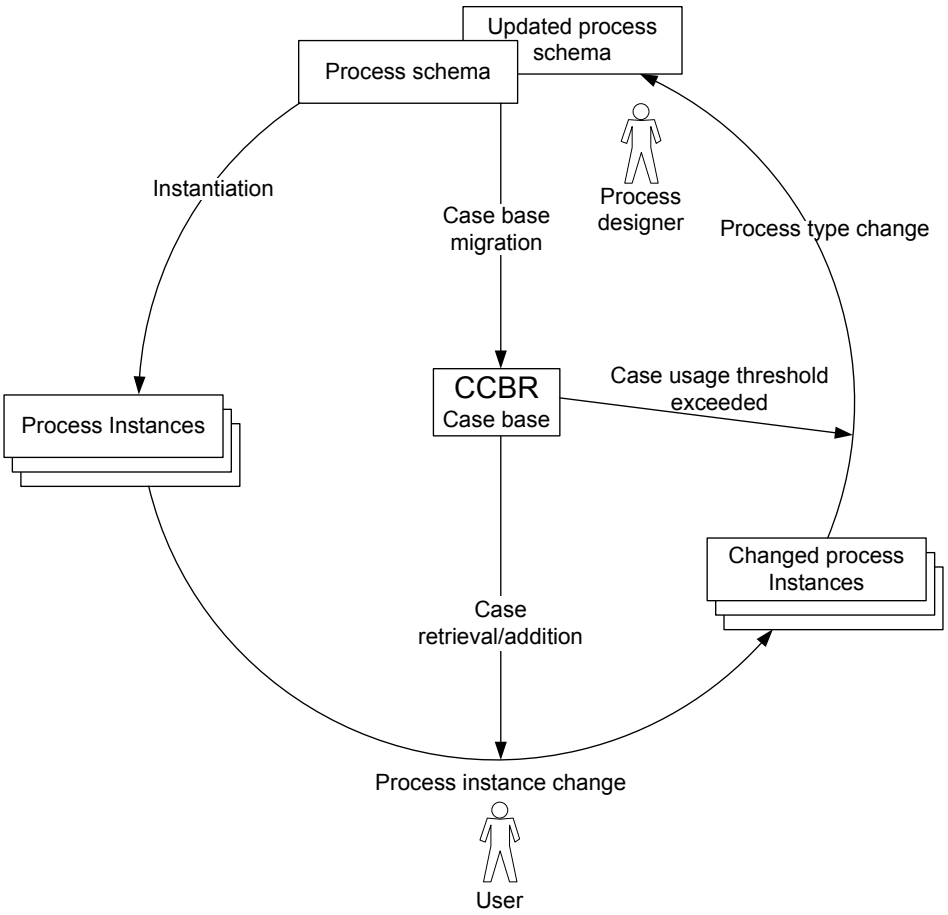


Figure 3.10.: Process life cycle of the integrative PAIS approach (compare [WRWR05] Figure 2)

migrated to the new schema. Due to both system's characteristics, correct and consistent ad-hoc modification can be assumed along with a memorization of changes and adaptive process types.

An extended CBR cycle (compare Figure 3.1 on page 68) is realized by PAIS in the following way: Upon the **addition of a new case** to the case base, a free textual *description* of the exception is given along with a set of question-answer pairs describing the exception's *reason*. A subset of the change operations made available by ADEPT and supplemented with parameters are available to process-creating users to specify the resolving *action* which is taken by the case. Notice that a retained case refers to a specific process schema version.

Case retrieval works in the same way as it did for CBRFlow: A dialog is initiated which consists of a set of questions and answers. The user's *answers* and additional *parameters* specify a matching on cases, which is refined by optionally given *operations* and a *subject*. Based on that information, the case base is filtered and a *similarity measure* similar to the one shown in Example 17 is used to present the best matches to

a requesting user.

Case reuse is assisted by case retrieval as described above. Change operations offered by ADEPT are used to revise the case. However their application requires some experience and is not in any case straightforward, because changes may imply additional *concomitant changes* in order to uphold correctness and consistency. The number of reuses is counted for each case.

Case evaluation as a part of case retainment is a feedback channel describing the usefulness of an applied case. A task containing a simple evaluation (positive/neutral/negative) and a descriptive text field is added at the end of a reused/retained case. This yields a ranking of reputation amongst cases and is displayed during retrieval. It helps finding the most successfully used cases in the past.

Case revision foots on the evaluation system and is invoked when a case receives negative feedback. This triggers process designers to either revise the case or to remove it from the case base.

As requirements evolve, exceptions show up more frequently. The **derivation of a process type change** is started when CBRFlow notifies that certain exception types are used very frequently. Process designers can react on this situation with a process type change. The notification is sent out as soon as a certain *threshold* of reuse frequency has been exceeded. However, type change induces a migration of running cases: ADEPT makes a distinction between *compliant* and *not compliant* instances. The former designates a class of instances on which the change can be applied. Not compliant changes can not be changed and the respective cases continue running on the old schema. Compliant cases can be either *biased* if they contain ad-hoc changes or *unbiased* if they do not. Unbiased cases are directly relinked to the new process schema, whereas biased cases require additional correctness checks.

3.3.4. Assessment of usefulness

Flexibility of process models and their instances is the *decisive* requirement for the runtime engine of Emergent Workflow if we leave elementary functionality like (RE1 - RE3 with respect to Table 2.4 in Section 2.3.10) aside.

The introduced work by Ellis et al. on page 90 considers schema evolution (RE4) and presents an approach to verify correctness for migrating instances to a new schema (RE7). On the underlying Petri Net formalism, immediate changes are potentially unsafe. That is why Ellis et al. propose to artificially delay the transition of running instances to the new schema by a construct named *synthetic cut-over change*. This assures for certain types of changes a consistent migration of Petri Nets in the middle of execution (RE7). Notice however that if this is put into practice and applied more often to a schema (which is surely the case for Emergent Workflow), it grows more and more "dead" branches. As a matter of fact, for iterative changes clean-up steps that cut the

old change regions off are mandatory in order to maintain a meaningful process model.

Next, ADEPT is presented on page 92 as a fully functional process management framework. Notice that only the ad-hoc change functionality of this framework is considered here [RD98](RE6). Fundamental for all aspects of ADEPT's properties are its *symmetrical control structures* which cause a highly rigid block structure. This is why its schematic elements are roughly outlined first. Among others, powerful control structures such as loops and failure edges are included in ADEPT which increase its expressive power, but induce also a more difficult handling. For instance the loop construct creates the necessity for an advanced change management and methods for undoing temporary changes, which complicate its handling. The most distinguishing feature however is at the same time the biggest benefit of ADEPT over other approaches: The application of its set of change operations does not shift the responsibility for correctness and consistency checking to a user or process designer, Instead, ADEPT is able to a assure both for most cases on-the-fly (RE7).

The last presented flexibility approach on page 96 is PAIS by Weber, Rinderle et al. Its decisive quality is that it integrates many features from two approaches which are already powerful by themselves: ADEPT mentioned above and CBRFlow. It offers the full range of functional features offered by both systems plus some synergetic effects. Schema evolution (RE5) with automatic consistency assurance for instance migration (RE7) and ad-hoc changability (RE6) are integrated in the CBR cycle. Conversational case-based reasoning allows the reuse of cases (RE5) and annotates them with descriptions (RE9). On top of that, the CBR cycle is extended with case evaluation functionality which improves the accuracy of individual cases.

4. Architectural proposal

"Grasp and reuse"

Emergent Workflow is aiming at a way to *grasp* the current procedures and processes and to *reuse* them in a way which is most simple, fast and flexible enough to be accepted by users. From a functional point of view, no single approach presented in Chapter 3 does cover all aspired aspects of Emergent Workflow. In order to receive the focused goals of Emergent Workflow, relevant ideas of different approaches have to be bundled and integrated into one system.

Not only technical issues are decisive factors, but in the first place users representing the *human factor* are. It may be emphasized at this point that the major motivation for Emergent Workflow is to overcome acceptance deficiencies that conventional workflow management systems are confronted with.

It appears recommendable to propose a *staged introduction* of an Emergent Workflow System for a number of reasons: First, the high complexity of a system implementing all kinds of desired functions and related technologies would be hard to implement, manage, administer and use for all involved groups – developers, process designers, administrators and users. Second, a step-by-step introduction is more likely to be accepted by users which is a crucial success factor for Emergent Workflow in particular. Finally the emergent approach implies that small-scale responsibility for process creation and planning is shifted to users. These are however demanding tasks that require knowledge, predictive thinking and not at last experience. Introduction phases allow users to slowly adjust to new procedures and give them the chance to get acquainted with new tools and to master their new tasks before they become mission-critical.

Creative activities are the most valuable and at the same time the most critical and fragile part of knowledge intensive work. Therefore a major amount of care is suggested when any kind of change is applied to them as their reaction to change is most sensitive. The introduction of information technology such as Emergent Workflow has a massive impact on the way creative processes function. That is why it is proposed that those components of Emergent Workflow that introduce the most profound changes are integrated at last.

The main objective of the architectural *first stage* is to gain user acceptance. It focuses rather on non-functional issues. Workflow technology is used at this stage in a way invisible to the user and *grasps* information. However, functional improvements and *reuse*

aspects are postponed to the next stage. The *second stage* brings functional changes into play: It introduces a workflow management system enriched with components for flexibility enhancement used for routine process parts only. Exception handling is here pioneering Emergent Workflow's ability to *reuse* past experience. The final *third stage* shifts controls and initiative to users. This offers a high potential to *reuse* past process fragments and the workflow becomes "emergent", but at the same time becomes also more complex to handle.

4.1. Stage 1 – Basic functionality

The goal of the introduction of basic functionality in the first stage is to obtain user acceptance for a minimum set of functions in the first place. A critical user should decrease rejective opinions by observing that the new system "actually does not harm" or even "helps a little". The idea is to keep as much familiarity of the user with tools he is used to instead of throwing him into a radically newly designed system environment. Automatizing some minor routine work should help decrease aversion.

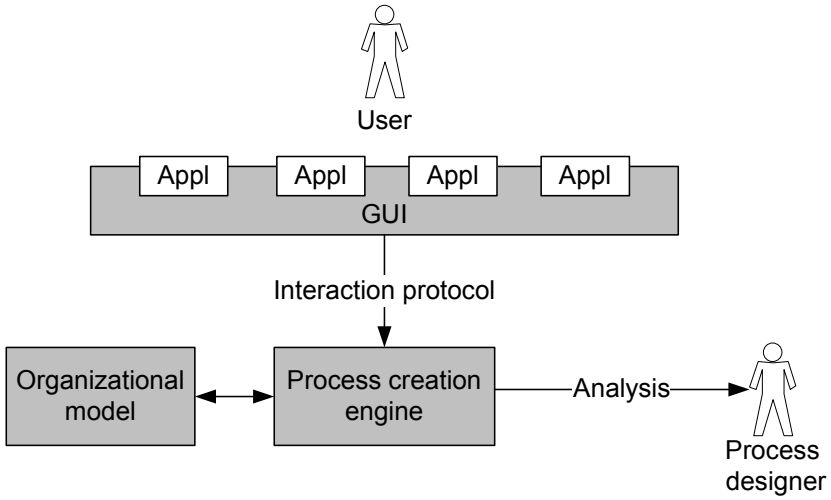


Figure 4.1.: Proposal stage 1

Figure 4.1 shows the arrangement of some basic workflow components introduced by the first stage. A Graphical User Interface (GUI) is presented to workflow users which integrates their applications into one common interface. The GUI creates a rough interaction protocol and forwards it to the process creation engine. Supported by an organizational model, it helps process designers to analyze the interactions of users.

Notice that at this stage, no formal workflow management is introduced regarding the process perspective (see Figure 2.14 on page 54). Workflow components have only passive, "observing" functionality for analysis and prepare next steps. That implies that

existing work patterns are not harmed or changed. Only behind the curtains – invisible for the user – changes and analysis take place in the form of process creation. The following paragraphs give more details on the function of components at this stage.

Integrated GUI A common interface that integrates most applications and tools is a cornerstone for building an Emergent Workflow. Functionally, this GUI must be able to create and output a protocol of user interactions. This includes basic information such as which application have been started or stopped. Additionally, it offers a generic interface to client applications who can plug into the common architecture. First, an extensible application interface helps to fill the user interaction stream with details on intra-application interaction. Combined, an interaction protocols tells what applications a user chose (e.g. started billing software) and what actions were performed inside the applications (e.g. chose customer order overview, edited the latest order, sent out bills). Second, the application interface facilitates the invocation of applications with parameters specifying a context, e.g. a customer ID. For certain roles, user defined variables determining a stateful GUI can automatically set application parameters, e.g. a part number set once is a parameter for all applications. Moreover, different groups of users employ different applications. Therefore, this interface must be tailored individually for each user type, e.g. by using authentication mechanisms. This suggests the creation of a basic organizational model: If users and their roles are known, then the interface can be composed of modular role-dependent elements such that a user receives an individually composed desktop. Finally, an abstraction from the operating system of a GUI is desirable in terms of look and feel. If the look and feel of an interface is easily configurable on each given platform, then future hard- and software changes will hardly affect users any more.

Organizational model With respect to the workflow perspectives (Figure 2.14 on page 54) the organizational perspective is the only perspective of a workflow management system which is visible for users at this stage. It is represented by an organizational model (see Figure 2.12 on page 2.12 for an example) that is used to abstract roles and groups from individuals. At this stage, the focus lies rather on role/user translation than on hierarchical relations between roles as it is mainly used for role-resolution by the GUI and the process creation engine.

Process creation engine A basic version of the process creation engine collects interaction protocols from users and offers basic data mining functionality. That includes an instance-based visualization of interaction protocols and elementary filtering options. Process designers rely on it to get an overview of the structure and types of individual users' activities. If privacy is an issue, anonymization of captured information by role abstraction helps to protect privacy and reduces user rejection and disapproval. It does neither focus on cross-role or departmental process relation nor does it present its outputs as feedback to users.

Metamodel As neither a real workflow management system nor a formal process representation is present, only a very limited workflow metamodel is needed at this stage and comprises two parts: First, the organizational model formalizes roughly the corporate hierarchy. Role abstraction by itself does not capture formal collaborative dependencies in the form of project or work groups. Second, the basic process creation engine is used by process designers to get an idea of the structure of each individual's tasks. Hence, a simple representation of process instances being fragments of a more complete process is necessary. That includes simple activities, basic control structures and no explicitly defined granularity level.

4.2. Stage 2 – Advanced functionality

The second stage provides advanced functionality and introduces a more complete workflow support and a reuse aspect while trying to maintaining simplicity. After users have gotten used to a new interface, now it is the goal to introduce functionality that creates a positive user experience like "This saves really time!" or "I had to type this only once!" At this stage, the desired result is the retainment of current work for later reuse without intermediate steps involving third parties such as process designers.

A workflow management system is introduced which is meant to support routine work, but not creative work. As it is known that the acceptance of a regular workflow management system would be too low due to its inflexibility and rigidity, two things are done: Support is restricted to rather static small-scale routine processes and deviations from routine are handled by a case base.

During the first stage, process designer were able to observe and analyze activities of users and to identify recurring routine tasks. Now designers create simple process model representations of these processes and offer them to users in the integration GUI. If deviations from common procedures occur, users can document the situation, apply instance-level changes and put the case into a case base for later reuse.

As Figure 4.2 shows, additional components extend the system introduced in stage 1. These new components focus on the support of *routine work*. Any kind of work (including creative work) is being captured by an interaction protocol, but only parts identified by process designers as routine work are further considered. In fact, by analyzing the interaction protocols, process designers choose static recurring processes, create a process model (not shown in the picture) and add it as a default case to the case base. Users performing routine work can (1) retrieve the default case and (2) enact it on the runtime engine. If one encounters an exceptional situation, it can be documented and ad-hoc changes inside the system are applied. After termination, (3) the case is retained and can be retrieved for later reuse. Documentation makes use of the dictionary by using existing or new keywords to describe the stored case. This description helps to find cases of a certain type during retrieval. Notice that the organizational model is here a

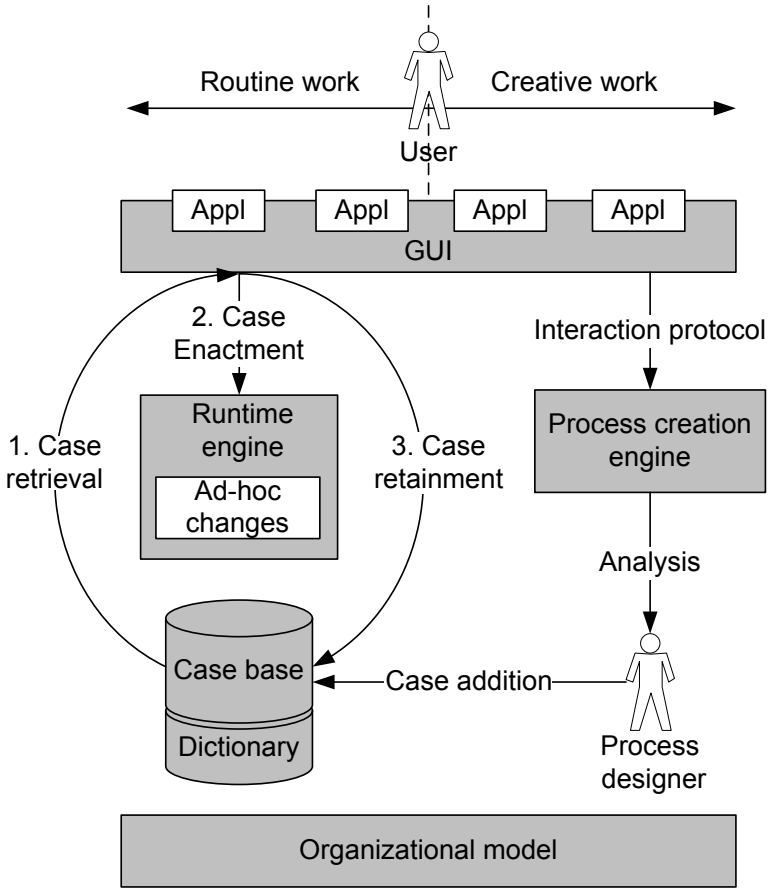


Figure 4.2.: Proposal stage 2

component commonly used by all other components – connecting edges are left out in Figure 4.2 for better readability.

Case base Initially, process designers populate the case base with *regular* cases of different types. As only routine work is put into cases, deviations from cases are usually caused by exceptional situation which have to be indicated explicitly by users. When entering a new case, the creator must provide a description characterizing the case. A case description answers for instance the following questions: When happened what type of exception and what is the compensation action? Was the exception compensated inside or outside the system? That information allows to classify types of exceptions. Compensation of exceptions can be taken care of inside the system by the ad-hoc adaption abilities of the runtime engine.

As cases inside the case base must be structured according to their type of exception, the description of a case does not only comprise free text, but also *keywords*. These keywords are defined in the *dictionary* attached to the case base. It represents a simple ontology of keywords that describes the attributes of all exceptions. Process designers

are supposed to initiate the dictionary, but users should be able to extend the dictionary with new terminology as they describe their case. This makes sure that user experience is immediately available for further reuse.

The goal is to populate a case base around established process types with numerous cases representing solutions for common exception types that were experienced in the past. When new exceptions show up, the case base is searched using descriptive keywords from the dictionary. That way, the organization, description and retrieval are highly related with each other. This bears the advantage that users actually are motivated to add accurate documentation to their processes. The better a case is described, the more likely it will be for a user to find and reuse a case later on.

Example 23. A minimum framework for a dictionary with key informations describing an exception scenario could be given as follows: *Who* (role, person, group) performed *what task* (type and instance) *when* (point of time, duration) and *what exception* (type: functional/nonfunctional, description) happened and was handled using what *compensation* (ad-hoc change operations inside the system/outside the system). Alternatively, the characteristics of exceptions as given in Section 3.1.2 on page 79 by Hwang et al. gives hints on relevant data types.

Ad-hoc instance adaption Ad-hoc adaption functionality on instances during runtime allows to compensate or resolve occurring exceptions inside the system. Therefore, individual users must be allowed to apply simple structural and state changes on running instances representing cases from the case base. A relatively small set of change constructs is made available to them in order to perform changes that are necessary for exception handling. They include the insertion or deletion of a task in sequence or parallel. It is recommended that the set of change operations offered to users is *minimal* (not more operations available than needed), but *not complete* (not every allowed instance structure can be reached from any given structure) with regard to the given workflow metamodel. As the default cases created by process designers might contain advanced control structures, the complexity of a complete set of change operations would very likely overwhelm common users. For example conditional forks and joins require the formulation of boolean expressions. Such tasks may introduce a level of complexity which is too high for unfamiliar users. This decision tries to realize treatment of exceptions inside the system by means that are straightforward enough to be applied by unexperienced users.

Metamodel In stage 2, a fully functional workflow management system is introduced. Therefore the organizational model from stage 1 is extended with metamodel constructs belonging to the *process perspective*. *Process types* are defined by process designers including structural elements to define the control flow between activities. In order to reduce complexity, no or a very simple data model (compare the *information perspective*) may be used. An instance of each process model resides as a *regular* case in the case

base and makes the process type accessible to users. Each instance put into the case base must be supplemented with a description consisting of dictionary-related keywords and some free text fields.

Basic change operations (insert, delete, both either sequential or parallel) on process instances are required to enable exception handling. As already mentioned in the previous paragraph, ad-hoc change operations available to users do not cover the complete metamodel used by process designers to create process types.

4.3. Stage 3 – Full functionality

The goal of the third stage is to provide support for all types of tasks. Here the key ideas of Emergent Workflow are made available for use with flexible tasks. After the user gets acquainted with the functionality, his experience should express something like "I don't know how we did our work before we had this system!". This means to create a system that formalizes processes relatively detailed without formalizing and complicating the view on them. A user performing creative work will typically receive a roughly structured bigger task assignment. He requires individual choice and freedom on the way the task is split up into single steps and accomplished. A supportive system offers, but does not force him to take a look at past executions of similar tasks and eventually adapt and reuse one of them. This stage tries to accomplish that by permanently monitoring the interactions a user. As he proceeds and requests support, the system may find similar patterns in previously recorded actions and proposes to reuse them. The user can then either agree to copy his previous procedures or decline and continue on his own.

It can be seen from Figure 4.3 that the previous stage's functionality is included and extended. The case-based reasoning cycle is still integrated and the numbering of its steps are prefixed with "C". What has been added is the ability for *schema evolution* of process types in the runtime engine. The dictionary has been extracted from the case base and forms together with the organizational model a component *general knowledge*. It is commonly used by all other components. The newly created cycle prefixed by an "F" represents the flow of fragments supporting the reuse of creative work patterns. First, the interaction protocol enters the runtime engine (F1) and is consolidated with server-side events (which are not shown in Figure 4.3) into an *audit trail*. Notice that the *time management component* as specified in Section 2.3.5 is considered an integral part of the runtime engine and is not shown in Figure 4.3 for better readability either. The process creation engine reads the audit trail (F2) and outputs process fragments. A user can signal in the interaction protocol that he completed a subtask. If, as a consequence, the workflow fragment represents a completed subtask, then the fragment is stored to a fragment base (F3b). Otherwise, an incomplete fragment is sent to the process matching engine (F3a) which compares it on-the-fly with stored existing fragments (F4). If sufficiently good matches are found, these are presented to the user who can select from the proposed fragments (F5) and reuse them by reenactment (F6). Apart from

that, process designers analyze the fragment base and the case base. On the one hand, they perform schema evolution on the process types in the case base if necessary and take care of the migration of running instances. On the other hand, they attempt to *compose* an overall process out of the process types, cases and stored fragments.

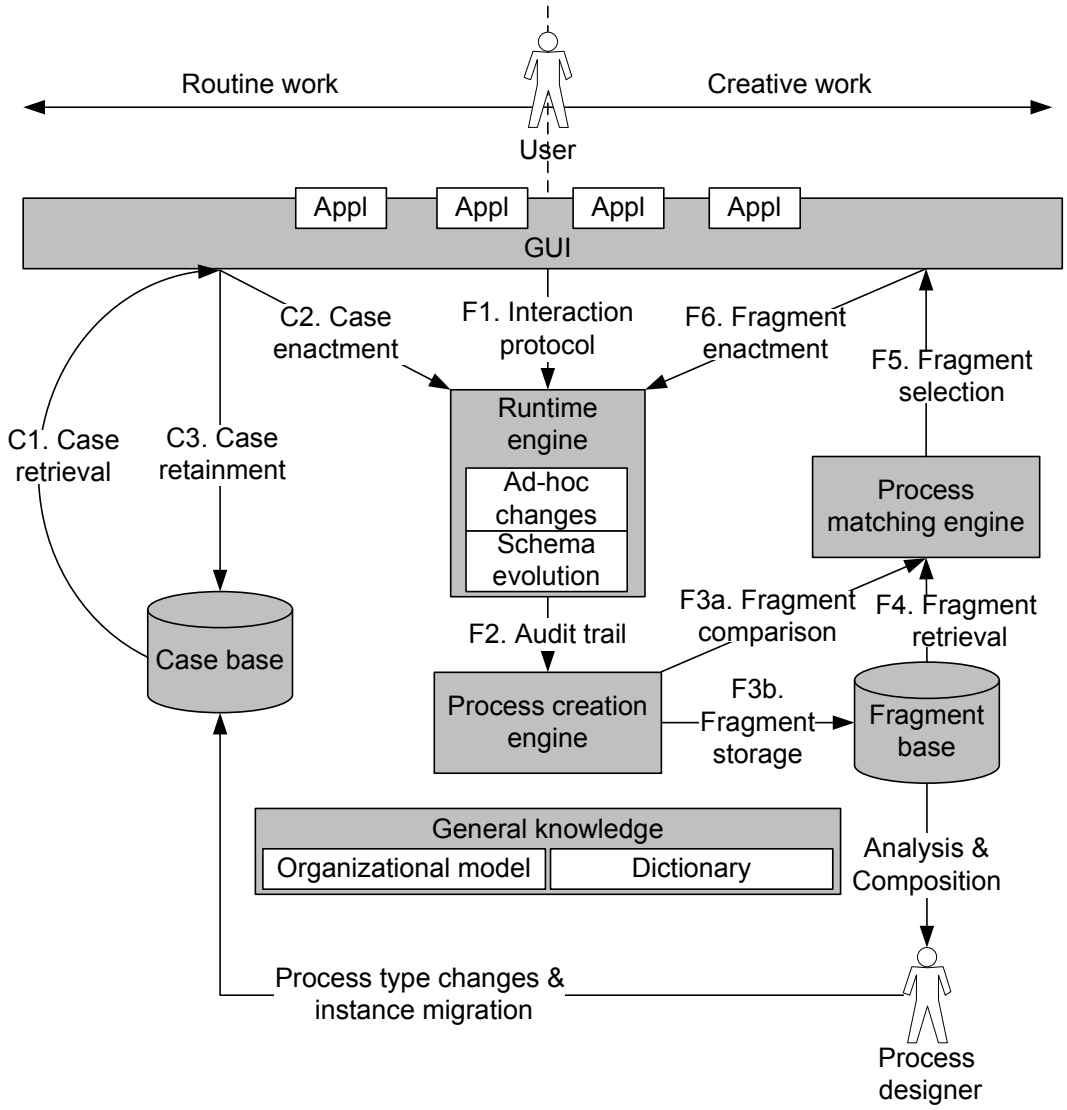


Figure 4.3.: Proposal stage 3

Schema evolution *Schema evolution* on the case base resembles the double-loop learning approach of Wargitsch et al. (compare Figure 3.4 on page 78): Workflow users participate in the incremental design of non-flexible, slowly evolving processes: They can create cases for exceptions and annotate their changes with keywords and descriptions. Users perform changes on a small scale basis as they modify only the process parts they work on. Process designers rely on these when they adapt routine process types to new

requirements. In fact, just as in the example approaches discussed in Section 3.1, process designers observe the growing case base. When exceptions get more frequent and show a strong bias towards a certain exception type, they can revise the existing process models according to the changed requirements and update the regular case of a certain process type. According to the chosen metamodel, running instances can be either migrated automatically or need to be handled manually. Notice that on this stage, the part of the system handling routine work shows strong functional similarities to the PAIS approach introduced in the related work Section 3.3.3 on page 96.

Process matching engine Captured interaction protocols is now not only an analysis tool for process designers, but is also used as a comparison and reuse tool for workflow users. A partial in-progress process instance needs to be matched with archived instances for criteria given by the user. If the user signals the invocation, a partial audit trail is translated into a process fragment. Due to the flexible nature of creative work, exact matches are highly unlikely. Therefore, the process matching engine needs to implement approximate search algorithms that compare syntax and structural similarity. Furthermore, an adjustable filtering threshold populates and limits the resulting list of matches. From there, the user checks manually on the results for semantically similar matches and eventually chooses one of them.

Analysis & Process composition A big danger at this stage is that individual users take control of the process at a small scale level and a general direction and overview passes out of focus. This is where analysis and process composition becomes a vital part of the Emergent Workflow Process – they allow to gather an overview of what is going on in order to make big scale adjustments where needed. Furthermore, user-based changes need to be consolidated as to avoid collision of incompatible changes.

Practically, analysis includes as a major step the creation of current process types followed by their composition. The *analysis* and derivation of process models from instance-based fragments offers post-hoc insights in characteristics of the real small-scale process: Where show instance of flexible processes commonalities? How do flexible processes evolve over time? Thereby it serves as a foundation for e.g. cross-departmental synchronization. *Composing* an overall process of derived process models yields a "big picture" that shows deficiencies which span over individual users' and groups' horizon. As each user participates with a different view on the process, hierarchical composition of fragments requires the consideration of each fragments granularity level. However, as flexible process fragments are likely to be very diverse, a common formal process type can also be very complex. It can yield a composition that looks like totally different instances were stitched together in parallel which does not help very much. In that case, a separate composition of instance fragment returns a view on the overall process of one single individual instance.

Metamodel The introduction further functionality at this stage requires an extension of the process metamodel. First, process types in the case base can now be changed schematically. That induces a set of change operations on process types. As the process type evolves now, its schema must be versioned in order to be clearly referrable. As ad-hoc alteration of cases is still allowed, instance changes apply to a certain schema version only. This leads to an ordered list of ad-hoc and schematic changes associated with and applied on each instantiated case from the case base. As already mentioned before, type change of running instances is not a trivial problem, therefore the use of a safe process metamodel as proposed by Weber, Rinderle et al. [WRWR05] is suggested.

Incomplete process fragments are created during run time and stored for later reenactment. They are based on a process model that allows the instantiation of incomplete models. For each activity, parameters changing during enactment (compare Section 2.4.1 pages 57 et sqq.) have to be reset to an initial state. The composition of fragments leads to compound instances which hold the same properties as their elements. For overlapping and especially hierarchical compositions, a measure for the relative level of detailedness of a fragment is introduced. It introduces *levels of granularity* according to hierarchical levels in the organizational model. As all events caused by user activities are captured, one can assume that each individual's interaction protocol presents a maximum level of detail. Therefore the position of a role determines the granularity of its recorded process.

5. Discussion

As many aspects of Emergent Workflow and its requirements were already discussed in Chapters 2 and 3, the following discussion will confine itself to issues which may arise from the integration of approaches presented in the architectural proposal, Chapter 4.

Functional issues of integration

Namely, Case-based reasoning (CBR), ad-hoc adaption of instances, schema evolution, process mining and process composition were introduced throughout this thesis. The following Table 5.1 couples approaches pairwise and assigns each pair a number. Referring to that enumeration, the following paragraphs discuss shortly the problems of coupling the two approaches, if they have not been discussed yet and are relevant.

	Ad-hoc adaption	Schema evolution	Process mining	Composition
CBR	(1)	(2)	(3)	(4)
Ad-hoc adaption		(5)	(6)	(7)
Schema evolution			(8)	(9)
Process mining				(10)

Table 5.1.: Composition of ideas

(1) CBR and Ad-hoc adaption As it has become evident in Section 3.1 on case-based reasoning, ad-hoc adaption can be made an integral component of CBR: Case revision implies the adaption of a retrieved case before its enactment. Ad-hoc adaption as a revision tool can actually extend the allowed revision phase to the complete enactment phase. That makes the CBR cycle more flexible and allows the ad-hoc adjustment of cases to any upcoming situation. However, the allowance of changed cases can also lead to a problem of classification: If a complete set of change operations is provided, theoretically one case can be modified in such a way that at the end it resembles more to another case than the originating case. This arises the question whether during case retainment, it should be stored as case of the first or the second case type. One can either choose to keep the case system rigid or flexible. The latter allows to change a case type during run time whereas the former keeps the case type static as soon as a case has been instantiated. A flexible case management must allow the re-classification of a case at any given time

before retainment. Static case management requires the establishment of common criteria for cases as within case types, structural correspondence of instances can not be guaranteed. Case classification needs to be based on parameters that remain unchanged by any ad-hoc adaption.

(2) CBR and Schema evolution The integration of CBR and schema evolution has been issued in the introduction of PAIS in Section 3.3.3.

(3) CBR and Process mining Integration of CBR and process mining is twofold: If cases are enacted and the events of execution are used for process mining, nothing special happens. That is not surprising, because after all, cases are during enactment regular process instances with a supplementary categorization. Without further engagement, it is however problematic to add instance-based mining outputs to a case base. The reason therefore lies in the fact that common audit trails do not indicate by default the association of a running instance with a case type. Actually case creation or retrieval happens before process enactment. Therefore, no documented activity inside a process instance indicates a case type.

There exist two possible resolutions for that matter: The first possibility is to merge case attributes of each instance into the audit trail. That makes it easily possible to identify the case affiliation of each single event. The alternative to this very verbose and redundant marking is to embrace case selection or retrieval for a new process instance as explicitly mentioned first task to the instance. That way, the runtime engine can execute the virtual "assignment" task and one single event identifying the case shows up in the audit trail.

(4) CBR and Composition Issues caused by the integration of CBR and fragment composition are strongly dependent on the way a case base is used: If it is used to model exceptions around a regular case which corresponds to the underlying process definition, then cases make no difference to a post-hoc composition of archived instances. See for that case **(7) Ad-hoc adaption and Composition**. Either way, composition has to deal with instances that deviate from the given process model in a rather unstructured way. The other application of case base is for processes that have no regular case but a couple of equivalent cases. Here, composition may be only able to compose cases of the same type as each case represents a distinctive sub-type of a process definition. Cross-case composition would result in syntactically inhomogeneous and semantically incorrect overall processes.

(5) Ad-hoc adaption and Schema evolution The roles and cooperative functions of ad-hoc instance adaption and schema evolution have been issued in Chapter 2, Sections 2.3.7 and 2.4.

(6) Ad-hoc adaption and Process mining Commonly, process mining does not recognize whether an event in the log was caused by a regular or by an exceptional activity. Consequently, changed instances containing adaptions that were caused

by exceptional circumstances are rated equivalent to those instances with a regular schema and course of state transitions. Without further consideration, this would lead to the creation of wrong process types. In order to resolve this situation, a differentiation of regular and changed/exceptional events is required to enable process mining to recognize the status of events. Having this knowledge, policies that handle event types with different weights can be put into practice. The desired outcome is a process model whose structure is influenced by the level of importance of its contained events. Notice that this idea is comparable to mining algorithms that use stochastic models and frequency tables to filter out noise from the audit trail. Here, varying types of events have to be recognized, classified and are integrated into a preliminary process model. Finally, only those events exceeding a minimum weight threshold are integrated into the process model.

(7) Ad-hoc adaption and Composition Ad-hoc changes are applied to fragments of the overall process by each user individually. When composing fragments, these numerous views are aggregated into the overall process. Let now an exception occur at a certain point in the process. Multiple users with different perspectives on the process will encounter the exception. The composition of fragments contributed by them will be overlapping at the exceptional point. The question here is whether all of them interpret the situation in the same way. If they do, then all of them will apply comparable ad-hoc adaptions that resolve the situation from their point of view. In that case, fragment composition should work flawlessly. If however, each user develops a different understanding of the situation, then people will take compensating actions that do match semantically. As a result, composition of fragments will either not yield a meaningful result or the difference are so strong that fragments collide already syntactically.

In order to enable successful combination of ad-hoc adaption and the composition of fragments, a minimum level of synchronization between users is required to allow a common understanding and adaption of such situations. Then overlapping fragments match a common problem and their composition delivers an overall understanding.

(8) Schema evolution and Process mining As the schema of a process definition evolves, process instances being enacted on different schema versions look different. As it has already been broadly discussed in Section 3.3.1, the migration of already running instances from an old to a new schema is a complicated issue. These kinds of instances also complicate the outcomes of process mining. The entries in an audit trail created by a migrated instance refer half to the old schema and half to the new schema. Consequently mining algorithms applied on events from migrated instances deliver *hybrid process fragments and types*. These kinds of outputs are not useful for purposes such as composition or reenactment.

Therefore, the runtime engine needs to indicate instance migration either by marking affected instances or creating a system event in the audit trail. This helps a

process mining application to recognize events from migrated instances and to ignore them.

(9) Schema evolution and composition What has been observed on intra-fragment scale in the previous paragraph **(8)** holds for the overall process when schema evolution is coupled with fragment composition. If process types are changed at any time, an *overall process* that started before the change and ended after it, contains potentially three types of archived instances: Those according to the *old* schema, *hybrid* instances containing both schemata and instances enacted on the *new* schema. Thus, one can consider it an *overall hybrid process*. From a use-oriented point of view, this kind of composition is highly interesting as it documents in detail how well the overall process handled the migration. Difficulties that showed up either before, after or during the schematic transition become evident from the composition.

The critical momentum can be seen in a situation where no hybrid instance exists but all instances either terminated before or started after the process type evolved. If fundamental difference were introduced by the type change, it might become hard to compose syntactically and semantically differing instances. Eventually, process designers need to insert an additional *transiting instance* which connect the gap between old and new instances.

(10) Process Mining and Composition Processes are composed vertically and horizontally: *Vertical* composition of fragments represents the alignment of different views and hierarchies related to a common part of the overall process. *Horizontal* integration relates causally dependent process parts either sequentially or in parallel.

As already mentioned in Section 4.3, vertical composition of process fragments is based on an explicit specification of their granularity (compare also Section 2.4). As multiple fragments yield multiple overlapping views on common process parts, hierarchical correlations have to be identified. Moreover, composition can only be accomplished with activities containing a sufficient amount of descriptive characteristics in order to detect equivalences. For horizontal composition, those matching activities represent the interfaces between the individual views. However, the more detailed activities on fragment is, the less likely it is to match them with other interfaces. Therefore, one needs to identify attributes which are *instance-specific*, but not *view-specific*, such as temporal constraints.

Example 24. In a quality gate-driven process, all engineers have to reach a certain development stage until a deadline (the quality gate), which was specifically given for this project (the overall instance). The activity "Delivery of results for the current quality gate" may be named differently for engineers in different disciplines and have differing related objects, roles, applications etc. – these attributes are *view-specific*. In the audit trail, the common interface events can be identified by for example the timestamps of their execution. They are all alike for each discipline, but *instance-specific* as the deadline was given for this project only.

6. Conclusion

6.1. Summary and conclusion

In the industry and many other fields of work, organizations have aligned their business according to processes. Workflow management systems offering technological support for processes bear in principle many advantages. However until today, these gains could not be widely realized as the traditional business process cycle tends to be too inflexible for many applications. As a result, knowledge and process awareness gets lost because users circumvent the workflow management system and resolve issues outside the system.

Emergent workflow has the vision to offer individual users immediate support without the need for pre-modeled processes. By capturing fragments of the real process, it aspires to gain user acceptance, improve reuse of work pattern and increase process transparency.

The contribution of this thesis to Emergent Workflow is a detailed requirements analysis, an introduction of related work, a conceptual proposal and a discussion of possible obstacles. The identification and specification of use cases, components, interfaces and a suitable process metamodel represents about half of this thesis. It identifies numerous functional and nonfunctional aspects in a structured manner. Moreover, selected related work is considered and assessed according to the outcomes of the preceding requirements analysis. The related work part surveys work on case-based reasoning, process mining and flexible workflow management and restricts its view on fundamentals and some interesting advanced work. The conceptual draft for the architecture of an Emergent Workflow Management System proposes a successive introduction of features with increasing functionality and complexity. As the proposal integrates ideas collected from various related work, a final discussion reflects on upcoming obstacles with integration and further practical aspects of the proposal.

As a conclusion on this work, we observe that for most of the requirements considered isolated from each other, practical approaches and partial solutions exist. From our point of view, the true challenge for Emergent Workflow is the integration of manifold ideas into one functional *and* usable system. Given the set of requirements, one is tempted to focus technical and functional aspects only and to forget that a resulting highly complex workflow management system does not solve the problems it was meant to overcome. Therefore, we would formulate as a maxim for further work on Emergent Workflow: "Try to accomplish as much as possible with as few as possible."

6.2. Omitted and future work

As this thesis we settled on a conceptual level, working on it resulted in covering a very broad range of involved topics. We have to say that the related work part mentioned herein represents a very incomplete and punctual view on the field of relevant related work. As a result, many topics with high relevance were either not mentioned at all or not treated adequately with respect to their importance. The following list mentions some topics worth of further investigation:

- **Access methods** for client applications on various levels of interactivity. They enable workflow users to make use of collections of fragments, process types, cases and terminology.
- **Process designer applications** used for fragment analysis and composition.
- **Design of a dictionary ontology and organizational model** with respect to their creation, maintenance and usability.
- **Workflow security** that manages user allowance to access, modify, create and extend any types of data.
- **Transaction support** of business processes including suitable constructs and execution models.
- **Inter-workflow coordination** to allow an integration of the Emergent Workflow approach with other types of workflow management systems.

As this thesis has an introducing character on Emergent Workflow at best, future work on this topic is manifold. Based on the given conceptual architectural proposal, a more concrete architectural specification has to follow. That starts with conceptual decisions based on the given requirements: From the available approaches, those may be chosen which show best functional and integrative abilities. Then more specific questions regarding algorithms, protocols, ontologies and storage issues have to be answered. As by now, the final step would be marked by a prototypical implementation.

A. Supplementary Listings and Figures

A.1. CODAW

A.1.1. Process data model

```
1  <?xml version="1.0"?><!DOCTYPE WorkflowSchema []>
2  <WSID> WS2</WSID>
3  <WSName> Market-Pull Workflow </WSName>
4  <WSType> ProductDevelopment</WSType>
5  <WSDesc> A product development process for a new chip </WSDesc>
6  <TaskList> (Project_Selection, Product_Definition,... ) </TaskList>
7  <ComponentWorkflowsUnModified> WS21 </ComponentWorkflowsUnModified>
8  <ComponentWorkflowsModified> Null </ComponentWorkflowsModified>
9  <WorkflowInstances> (WFIns1 WFIns22 WFIns23) </WorkflowInstances>
10 <WFFormalModel>
11 <PNModel>
12 PN-WS2
13 </PNModel>
14 </WFFormalModel>
15 <Tasks>
16 <Task>
17 <TaskType> Business </TaskType>
18 <TaskName> Project_Selection</TaskName>
19 <TaskDesc> Selects a list of new product ideas to work on </TaskDesc>
20 <TaskID> 1 </TaskID>
21 <TaskDesign>
22 <Parameters>
23 <Param> ?project_list </Param>
24 <Param> ?total_budget</Param>
25 <Param> ?resource_list</Param>
26 </Parameters>
27 <PreConditions>
28 <Predicate> (available ?project_list) </Predicate>
29 <Predicate> (available ?resource_list) </Predicate>
30 </PreConditions>
31 <PostEffects>
```

```
<Effect> (add (new_proj_list ?new_list)) </Effect>
<Effect> (add (new_budget ?new_budget)) </Effect>
</PostEffects>
<SubWF> WS25 - A subprocess
</SubWF>
</TaskDesign>
<TaskFormal>
<FSP> PS = ( init -> sort_by_cost -> review -> vote -> select </FSP>
</TaskFormal>
<TaskDefn>
<Agent> General_Manager </Agent>
<Agent> Marketing </Agent>
<Agent> Engg_Design </Agent>
<Agent> Manfg </Agent>
<Agent> QA </Agent>
<Agent>Purchasing</Agent>
<Agent> Customer_Service</Agent>
<Procedure>
<ProcedureName> Select_Project </ProcedureName>
<ProcedureSource> HandBook </ProcedureSource>
<Implementation_type> Manual_Team_Execution </Implementation_type>
</Procedure>
<Inputs>
<DataItem> budget </DataItem>
<DataItem> resources </DataItem>
<DataItem> projects </DataItem>
</Inputs>
<Outputs>
<DataItem> selected_projects </DataItem>
<DataItem> remaining_budget </DataItem>
</Outputs>
</TaskDefn>
</Task>
```

A.1.2. Instance level workflow schema

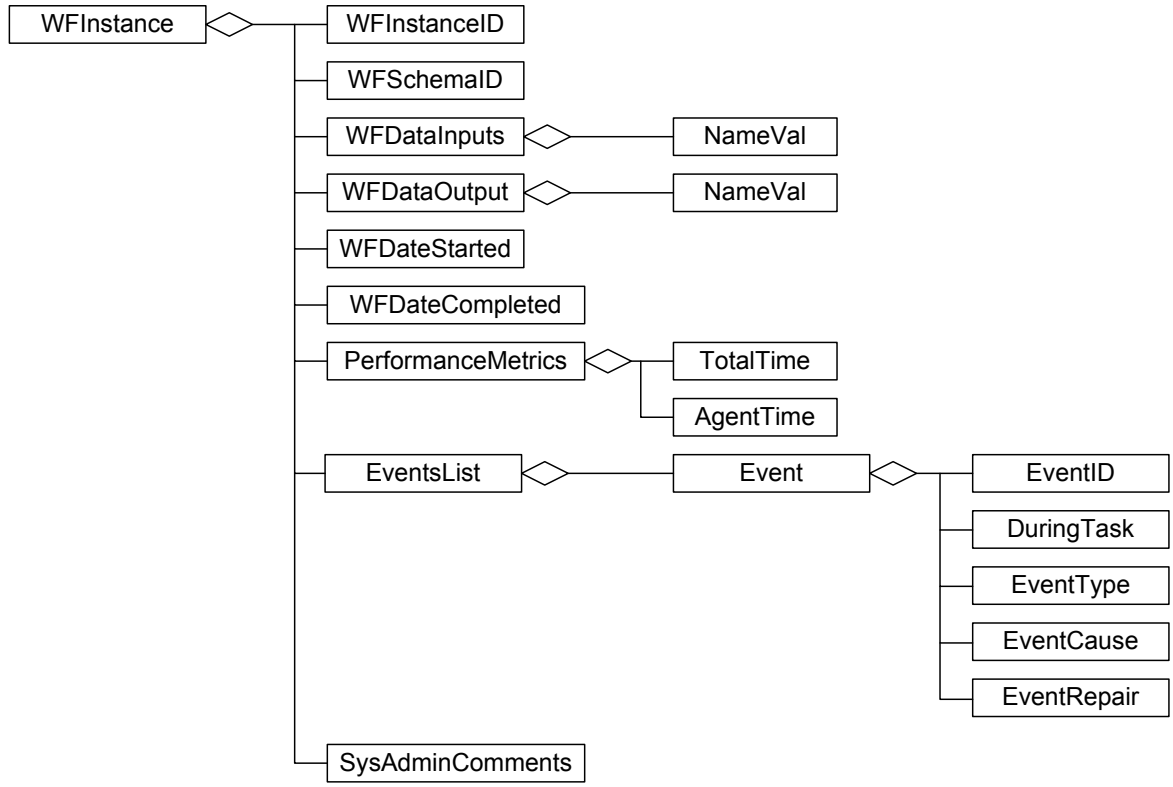


Figure A.1.: CODAW instance schema (compare [MZ03] Figure 7)

Bibliography

[Aal02] W.M.P. van der Aalst. Business Process Management: A personal view, 2002.

[ADH+03] W. M. P. van der Aalst, B. F. van Dongen, J. Herbst, L. Marustera, G. Schimm, and A. J. M. M. Weijters. Workflow mining: A survey of issues and approaches. *Data & Knowledge Engineering, Volume 47, Issue 2 , November 2003*, pages 237–267, November 2003.

[AH02] W.M.P. van der Aalst and K.M. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT press, Cambridge, MA, 2002.

[AHKB02] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow Patterns. QUT Technical report, FIT-TR-2002-02, Queensland University of Technology, Brisbane, 2002. (See also http://www.tm.tue.nl/it/research/patterns.).

[AJ00] W.M.P. van der Aalst and S. Jablonski. Dealing with Workflow Change: Identification of Issues and Solutions. *International Journal of Computer Systems, Science, and Engineering*, 15(5):267–276, 2000.

[AP94] Agnar Aamodt and Enric Plaza. Case-based reasoning: foundational issues, methodological variations, and system approaches. *AI Communications*, 7(1):39–59, 1994.

[AWM03] W.M.P. van der Aalst, A. Weijters, and L. Maruster. Workflow mining: Discovering process models from event logs, 2003.

[Bus01] Christoph Bussler. The Role of B2B Protocols in Inter-Enterprise Process Execution. *Lecture Notes in Computer Science*, 2193:16–34, 2001.

[CCPP96] Fabio Casati, Stefano Ceri, Barbara Pernici, and Guiseppe Pozzi. Workflow Evolution. In *ER '96: Proceedings of the 15th International Conference on Conceptual Modeling*, pages 438–455, London, UK, 1996. Springer-Verlag.

[DA04] Boudewijn F. van Dongen and Wil M. P. van der Aalst. Multi-phase Process Mining: Building Instance Graphs. In *ER*, pages 362–376, 2004.

[DFAB98] Alan Dix, Janet Finlay, Gregory Abowd, and Russell Beale. *Human-Computer Interaction 2nd Edition*. Prentice Hall, 1998.

[DS90] T. H. Davenport and J. E. Short. The new industrial engineering: Information technology and business process redesign. *Sloan Management Review*, pages 11–27, Summer 1990.

[EKR95] Clarence Ellis, Karim Keddara, and Grzegorz Rozenberg. Dynamic Change Within Workflow Systems. In *COCS '95: Proceedings of conference on Organizational computing systems*, pages 10–21, New York, NY, USA, 1995. ACM Press.

[GAHL01] P. Grefen, K. Aberer, Y. Hoffner, and H. Ludwig. CrossFlow: Cross-organizational Workflow Management in Dynamic Virtual Enterprises. *International Journal of Computer Systems, Science, and Engineering*, 15(5):277–290, 2001.

[HHJHS97] J. Hagemeyer, T. Herrmann, K. Just-Hahn, and R. Striemer. Flexibilität bei Workflow-Management-Systemen. In *Usability Engineering: Integration von Mensch-Computer-Interaktionen und Software-Entwicklung, Fachtagung Software-Ergonomie 1997, Dresden, 3.-6.3.97, Stuttgart*, pages 179 – 190. Teubner, 1997.

[HHT99] San-Yih Hwang, Sun-Fa Ho, and Jian Tang. Mining Exception Instances to Facilitate Workflow Exception Handling. In *DASFAA*, pages 45–52, 1999.

[Hol95] David Hollingsworth. Workflow Management Coalition Specification. The Workflow Reference Model, January 1995. Document Status - Issue 1.1.

[HSW97] Thomas Herrmann, August-Wilhelm Scheer, and Herbert Weber. *Verbesserung von Geschäftsprozessen mit flexiblen Workflow-Management-Systemen 1.* Physica-Verlag, 1997.

[JB96] Stefan Jablonski and Christoph Bussler. *Workflow Management – Modeling Concepts, Architecture and Implementation.* International Thomson Computer Press, 1996.

[KR93] James E. Kurose and Keith W. Ross. *Computer Networking. A Top-Down Approach Featuring the Internet*, volume Second Edition. Addison-Wesley, 2993.

[LS97] K. Lei and M. Singh. A Comparison of Workflow Metamodels, 1997.

[MAW03] A.K.A. de Medeiros, W.M.P. van der Aalst, and A.J.M.M. Weijters. Workflow Mining: Current Status and Future Directions. In R. Meersman et al., editor, *CoopIS/DOA/ODBASE 2003*, volume LNCS 2888, pages 389 – 406. Springer-Verlag Berlin Heidelberg, 2003.

[MGMR02] Sergey Melnik, Hector Garcia-Molina, and Erhard Rahm. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In *ICDE*, pages 117–128, 2002.

[MGR02] S. Melnik, H. Garcia-Molina, and E. Rahm. Similarity Flooding: A Versatile Graph Matching Algorithm and its Application to Schema Matching. In *Proc. 18th ICDE*, San Jose, CA, February 2002.

[Müh96] Michael zur Mühlen. Der Lösungsbeitrag von Metamodellen und Kontrollflußprimitiven beim Vergleich von Workflowmanagementsystemen. Master's thesis, Westfälische Wilhelms-Universität Münster, September 1996.

[MZ03] Therani Madhusudan and J. Leon Zhao. A Case-Based Framework for Workflow Model Management. Springer-Verlag Berlin Heidelberg, 2003.

[MZM04] Therani Madhusudan, J. Leon Zhao, and Byron Marshall. A case-based reasoning framework for workflow model management. *Data Knowl. Eng.*, 50(1):87–115, 2004.

[RD98] Manfred Reichert and Peter Dadam. Adept$_{flex}$-supporting dynamic changes of workflows without losing control. *J. Intell. Inf. Syst.*, 10(2):93–129, 1998.

[Rei00] Manfred Reichert. *Dynamische Ablaufänderungen in Workflow-Management-Systemen.* PhD thesis, Universität Ulm, 2000.

[RRD02] Stefanie Rinderle, Manfred Reichert, and Peter Dadam. Effiziente Verträglichkeitsprüfung und automatische Migration von Workflow-Instanzen bei der Evolution von Workflow-Schemata. *Inform., Forsch. Entwickl.*, 17(4):177–197, 2002.

[RRD04] Stefanie Rinderle, Manfred Reichert, and Peter Dadam. Correctness criteria for dynamic changes in workflow systems - a survey. *Data Knowl. Eng.*, 50(1):9–34, 2004.

[RWRW05] S. Rinderle, B. Weber, M. Reichert, and W. Wild. Integrating Process Learning and Process Evolution - A Semantics Based Approach. submitted for publication., 2005.

[Shn98] Ben Shneiderman. *Designing the User Interface 3rd Edition.* Addison-Wesley, 1998.

[SM95] D. M. Strong and S. M. Miller. Exceptions and exception handling in computerized information processes. *ACM Transactions on Information Systems*, 13(2):206–233, 1995.

[Ver04] Verein Deutscher Ingenieure, editor. *VDI 2206. Entwicklungsmethodik für mechatronische Systeme - Design methodologies for mechatronic systems.* VDI-Gesellschaft Entwicklung Konstruktion Vertrieb, June 2004.

[Wor99] Workflow Management Coalition. Terminology & Glossary, 1999. Document Number WFMC-TC-1011.

[WRWR05] Barbara Weber, Stefanie Rinderle, Werner Wild, and Manfred Reichert. CCBR-Driven Business Process Evolution. In *Proc. 6th Int'l Conf. on Case-Based Reasoning (ICCBR'05) (accepted for publication)*, Chicago, IL, August 2005.

[WWB04] B. Weber, W. Werner, and R. Breu. CCBR–enabled adaptive workflow management. In *Proc. European Conf. on Case-Based Reasoning (ECCBR'04)*, LNCS 3155, Madrid, 2004.

[WWT97] Christoph Wargitsch, Thorsten Wewers, and Felix Theisinger. Workbrain: Merging Organizational Memory and Workflow Management Systems. In *Proceedings on 21st Annual German Conference on AI '97*, 1997.

[WWT98] Christoph Wargitsch, Thorsten Wewers, and Felix Theisinger. An Organizational-Memory-Based Approach for an Evolutionary Workflow Management System - Concepts and Implementation. In *HICSS '98: Proceedings of the Thirty-First Annual Hawaii International Conference on System Sciences-Volume 1*, page 174. IEEE Computer Society, 1998.

Erklärung

Name: Florian Bertele
Matrikelnummer: 463675

Ich erkläre, dass ich diese Diplomarbeit selbst verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den 29. April 2005