ulm university universität
uulm

Ulm University
89081 Ulm
Germany

Faculty of Engineering, Computer Science and Psychology
Institute of Databases and Information Systems

Advanced Concepts for Task List Lifecycle Support

Master's Thesis at Ulm University

Submitted by:

Simon Stöferle
simon.stoeferle@uni-ulm.de

Reviewer: Prof. Dr. Manfred Reichert
Dr. Rüdiger Pryss

Supervisor: Nicolas Mundbrod

2018

Version from June 15, 2018

Satz: PDF-LaTeX 2$_\varepsilon$

Abstract

Globalization and the shift towards a knowledge-based society have increased the importance of knowledge work and in particular knowledge-intensive processes (KiPs) in highly developed countries. As a result, knowledge workers demand suitable information systems supporting their collaboration in knowledge-intensive processes. However, due to their difficult characteristics, there is still no such adequate support for KiPs. A KiP-supporting system needs to provide digital, lifecycle-based task lists to ensure sustainable support. Today, knowledge workers usually organize and manage their collaborative work in a KiP using paper-based task lists, e.g. to-do lists or checklists. Although task lists are intuitive and widely used, their current implementations tend to be ineffective and error-prone. Task lists are neither synchronized nor accessible by several knowledge workers simultaneously. In addition, no task list lifecycle support is provided and media disruptions aggravate task management. As a consequence, the efforts of knowledge workers in task management are not exploited for the optimization of future KiPs.

As part of the proCollab research project, this thesis addresses advanced concepts to support the task list lifecycle. For this purpose, existing lifecycle concepts are adapted and improved in particular. To allow an adequate comparison of task lists, an approach for a similarity analysis, on which the advanced concepts are based, is proposed first. As it is not always possible to create a suitable task list in advance, an approach for the automatic generation of a task list template from completed task list instances is presented. Furthermore, an approach for optimizing existing task list templates by incorporating the most frequent changes applied to task lists in use is explained. An additional approach for analyzing and identifying nested insert operations is proposed to extend and improve the optimization of existing task list templates. The presented concepts are together implemented in the current proCollab proof-of-concept prototype to demonstrate their feasibility and applicability. Therefore, various services and a central REST interface as well as a comprehensive test framework are implemented.

Contents

1 Introduction

As a result of globalization, society in highly developed countries is shifting more and more from an industrial to a knowledge-based one. Knowledge work has become the predominant type of work [31]. Therefore, it is of great importance to adequately support knowledge-intensive processes (KiPs) and knowledge workers (e.g. engineers, researchers or physicians). In areas such as research, healthcare and engineering, knowledge workers collaborate to achieve a common goal, such as the treatment of a patient or the development of a new car.

1.1 Problem Statement

However, the cooperation of knowledge workers in KiPs is difficult to support using traditional information systems. Driven by the knowledge workers at run time, KiPs are *emergent* and *unpredictable*, whereas traditional standardized processes expose a pre-defined structure. As a result, traditional approaches supporting standardized processes are not suitable for supporting KiPs, as they have not been developed to cope with continous change. In addition, several knowledge workers from different domains usually work together in a KiP to achieve a common goal. To achieve this common goal effectively, they need to communicate and coordinate with each other. For this reason, knowledge workers primarily rely on task lists (e.g. to-do lists or checklists) to organize and distribute tasks. Traditionally, paper-based task lists are still widely used and, therefore, not synchronized and not centrally managed. These issues make it difficult for knowledge workers to effectively work together. With the use of an information system supporting digital task list management, the core problems of paper-based task lists may be solved,

as the information system ensures the synchronization and availability of task lists. By realizing a lifecycle for digital task lists, these can be continuously advanced and improved. However, there is no adequate information system using lifecycle-based, digital task list management to provide adequate support for KiPs.
After knowledge workers have finished their work in a KiP, task lists are usually archived or even disposed resulting in a loss of knowledge and experience. An analysis of the completed task lists would make it possible to derive optimizations for future KiPs and to preserve knowledge.

Due to these problems, there is a strong demand for a new type of information system that systematically and sustainably supports knowledge workers within KiPs [25]. In particular, it shall ensure that tasks are always synchronized and accessible to all involved knowledge workers and media disruptions are avoided to increase the productivity of knowledge workers. With respect to a lifecycle-based task management, an automatic analysis and optimization of task lists shall be integrated to reuse and preserve gained knowledge and experience.
The proCollab research project conducted at Ulm University aims to establish this type of information system for the appropriate support of KiPs based on digital task list management and a lifecycle approach. proCollab relies on the key entities *processes*, *task trees*, and *tasks* to realize a framework for representing KiPs and the task-based artifacts used by knowledge workers for properly coordinating their work (cf. Figure 1.1). To enable a lifecycle-based task list management for KiPs, proCollab processes and task trees are refined to *process templates* and *process instances* as well as *task tree templates* (with *task templates*) and *task tree instances* (with *task instances*), respectively. In the course of the task management lifecycle realized in proCollab, a particular task tree template may be instantiated by knowledge workers to retrieve a task tree instance incorporating best practices and preserved domain-specific knowledge. However, there is no analysis and optimization phase that completes the lifecycle by analyzing the changes made to the task list instances by knowledge workers while working in a KiP. As a result, the changes made are not utilized for optimizing an existing task list template and, therefore, gained knowledge is not reused.

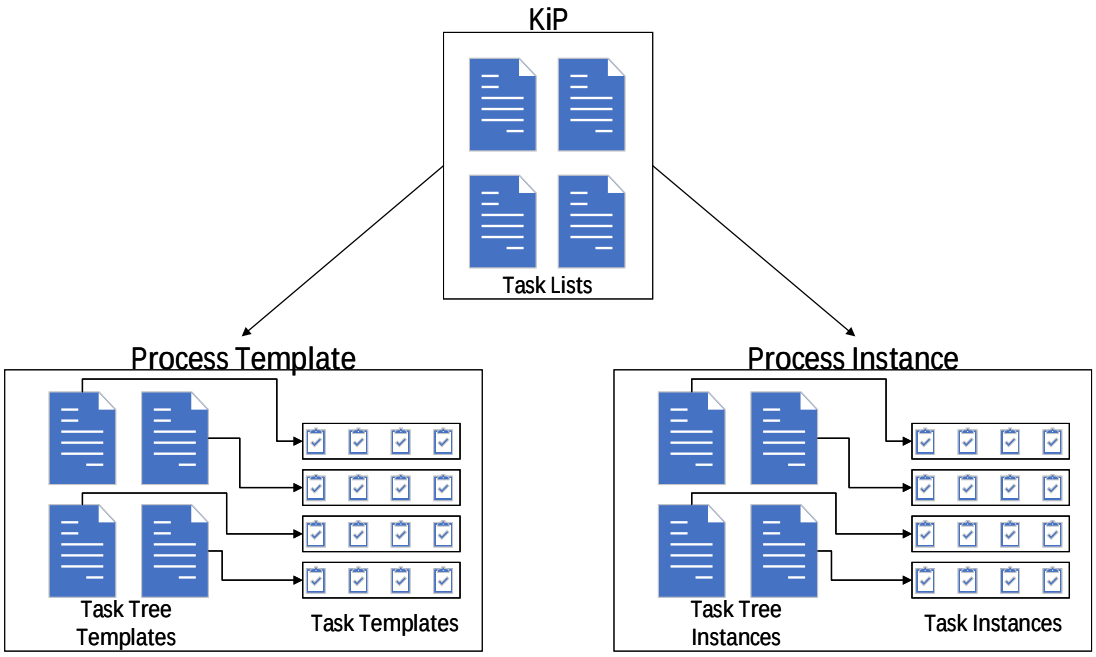


Figure 1.1: proCollab Components

1.2 Contribution

As part of the proCollab research project, a sophisticated proof-of-concept prototype has been developed demonstrating the feasibility and potential of the proCollab approach. By using digital task lists, knowledge workers may simultaneously access and manage tasks at any time and thus collaborate with other knowledge workers without consistency and synchronization problems.

Due to the constant *uncertainty* and the resulting *emergence* of KiPs, it is extremely difficult to provide knowledge workers with precisely fitting templates at all times. For this reason, this thesis deals with the implementation and advancement of existing lifecycle concepts for process and task management. Therefore, this thesis is based on Chen Li's PhD thesis [19] and Florian Beuter's master's thesis [3]. In [19] the *MinADEPT algorithm* was introduced, which represents an approach to generate a business process model out of a set of business process variants through the use of a cluster technique. The MinADEPT algorithm was adapted to accept task list instances as input in order to auto-

matically build a new task list template by clustering blocks [3]. This approach is called *cluster mining* approach.

Additionally, an approach for the automatic optimization of task list templates based on changes applied to task list instances was presented [3]. By analyzing the change operations performed on task list instances, the most frequent changes are identified and then applied to the original task list template. Consequently, this approach is called *change mining* approach.

In the course of this thesis, the cluster mining and change mining approaches for generating new or optimizing existing task list templates are adapted to the advanced data model of the new proCollab prototype. Furthermore, both approaches are improved by eliminating the weak points from the previous implementation. However, this was not possible without far-reaching changes and adjustments.

The cluster mining and change mining approaches facilitate to realize a task management lifecycle (cf. Figure 1.2). Knowledge workers may create a suitable task list template in advance. Before collaborating in a KiP, knowledge workers may select an existing task list template, or if there is no matching task list template, start the collaboration in the KiP with a task list instance created from scratch. In case an existing task list template is selected, it can be optimized after the KiP has been completed using change mining by analyzing the completed task list instances from similar KiPs and deriving optimization options from them. In the other case, i.e. if no suitable task list template exists, a new task list template can be generated from the various completed task list instances from similar KiPs by using cluster mining. An extension of the task list template optimization allows the identification of inserted sub-trees using the cluster mining algorithm (cf. Figure 1.3). The identified sub-trees may then be used to optimize the existing task list template.

Changes to task list instances may be necessary at any time while during the collaboration of knowledge workers in a KiP due to external influences. In order for these changes to be analyzed and evaluated in the course of change mining, it is necessary to log them. For this purpose, a logging concept is developed and implemented as part of this thesis to log operations on processes, task lists and tasks. The logging approach allows the creation of change and execution logs for task list instances that serve as a basis for analyzing change operations.

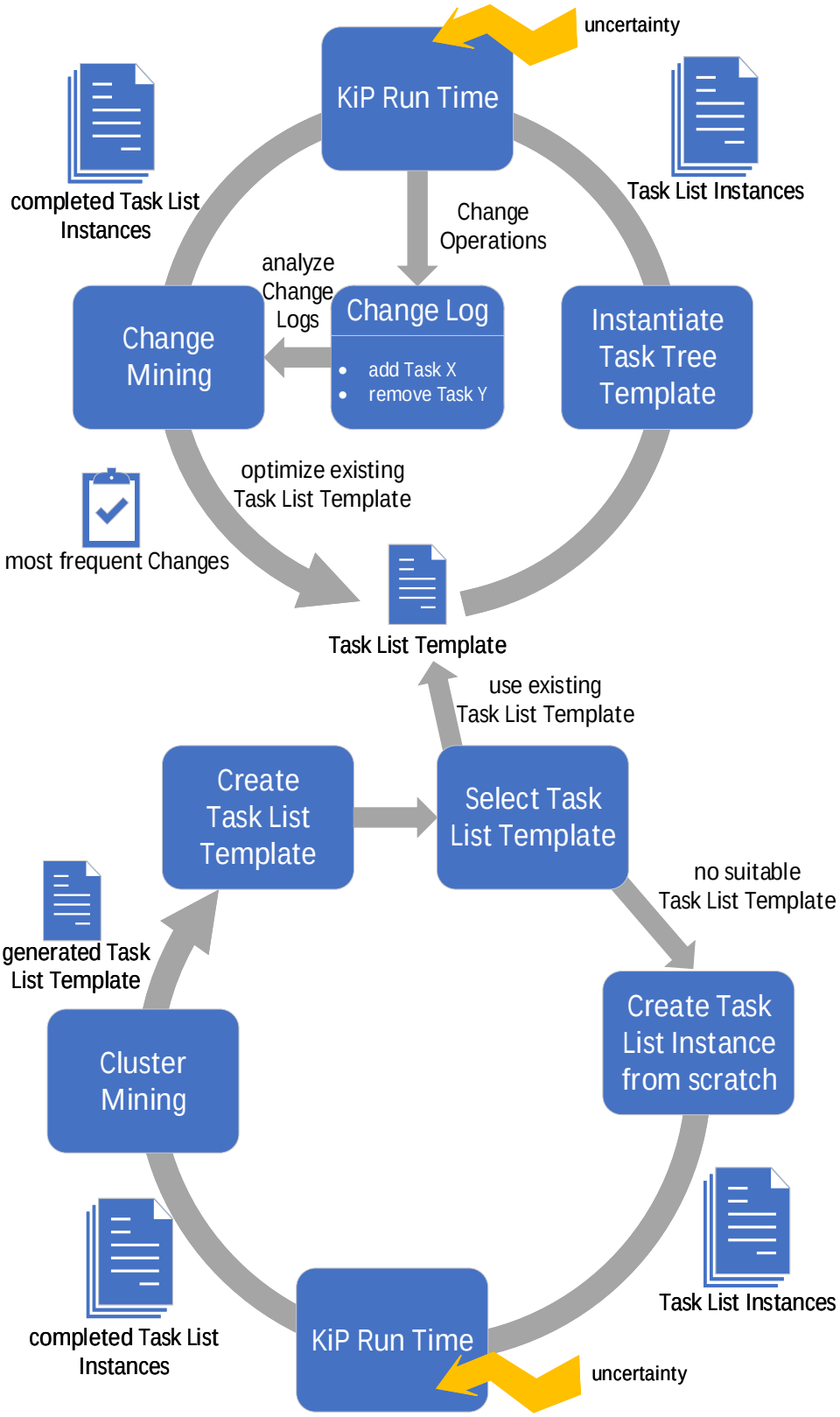


Figure 1.2: Cluster Mining and Change Mining Overview

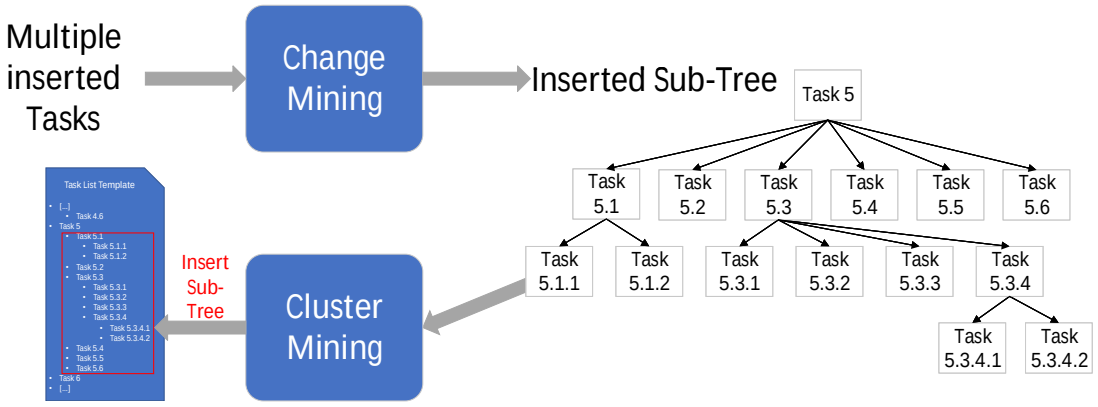


Figure 1.3: Inserted Sub-Tree

In preparation for the generation and evolution of templates, it is necessary to identify identical or similar tasks (or change operations) so that they are considered as such in the analysis. Therefore, this thesis proposes and implements an approach for a similarity analysis for tasks (or change operations on tasks) based on the corresponding entities and additional meta information retrieved from log entries.

The lifecycle services are implemented as part of this master's thesis so that knowledge workers can start the generation and optimization of templates in the desired manner. Four services are implemented:

- the central *Lifecycle Service*, which provides all operations for generating and optimizing task list templates by combining the corresponding subordinate services,
- the *Task List Generation Service*, which enables the generation of task list templates,
- the *Task List Evolution Service*, which enables task list templates to be optimized, and
- the *Similarity Service*, which performs similarity analysis of tasks and change operations.

To support knowledge workers with easy access to the methods of the various services, a central REST interface is implemented. The interface is then called by the proCollab client applications to invoke the operations of the lifecycle service.

Furthermore, a comprehensive test framework based on JUnit has also been implemented

to assess the various functionalities of task list template generation and optimization. For this purpose, different use cases are utilized to simulate the behaviour of knowledge workers, first creating either several task list instances (generation) or one task list template (optimization), depending on the test case.

1.3 Outline

The remainder of this thesis is structured as follows: Chapter 2 introduces the fundamental concepts this thesis is based on. Subsequently, the concept of a lifecycle-based task list management is presented in Chapter 3. Chapter 4 describes an approach to identify and group similar tasks as part of a similarity analysis and serves as a starting point for the further analyses. Chapter 5 covers the automatic generation of a task list template from a set of completed task list instances using cluster mining. Chapter 6 presents the optimization and evolution of existing templates through the application of a change mining algorithm. Finally, Chapter 7 concludes this thesis with a summary and an outlook on future work.

2

Fundamentals

This chapter provides the theoretical basis for the thesis by explaining the essential fundamentals and introducing the frequently used terminology. First, business processes and business process management are introduced. Next, the business process management lifecycle is explained. Subsequently, knowledge-intensive processes are discussed. Further, the proCollab research project is presented, followed by the discussion of the existing task list change operations. Finally, the concept of process mining is explained.

2.1 Business Processes

There is a variety of definitions for the term *business process*. Definition 2.1 introduced by [48] shall be used in the following.

Definition 2.1. *A **business process** consists of a set of activities that are performed in coordination in an organizational and technical environment. These activities jointly realize a business goal. Each business process is enacted by a single organization, but it may interact with business processes performed by other organizations.*

Furthermore, a *business process management system* (BPMS) is defined in the following way [48].

Definition 2.2. *A **business process management system** is a generic software system that is driven by explicit process representations to coordinate the enactment of business processes.*

To be more precise, a BPMS is deployed to support management and organisation of various business processes within a company. It enables the modeling of real world business processes in the form of *process templates*, which are stored in so-called *process repositories*. To obtain an appropriate process template, the corresponding business process is analyzed and documented first. A process template includes activities, events and decisions in such a way that it depicts a real world business process. Once a process template is specified for a business process, the future process occurrences can be supported by the BPMS by providing *process instances* through interpreting the process template. The BPMS is able to support the business process in a pre-defined, standardized manner as it uses the process template to determine the process instance's activities, their execution order and the assigned users. In summary, the deployment of a BPMS may reduce time delays, costs and error rates for the given business processes within a company.

Figure 2.1 shows an example business process represented in the Business Process Model and Notation (BPMN) notation. The business model describes how a simple order process may be defined and executed.

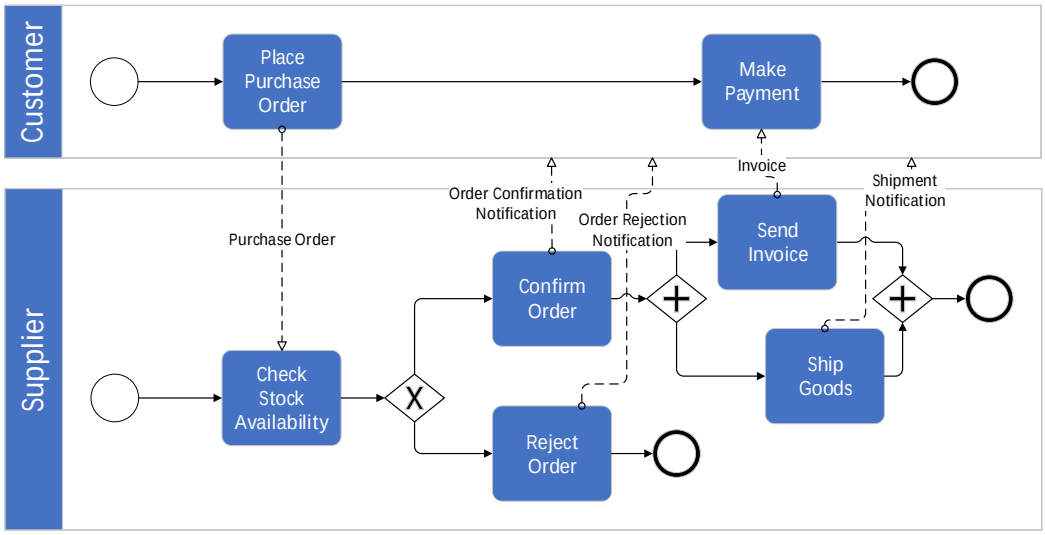


Figure 2.1: An Order Process in BPMN notation

The customer starts the process by placing a purchase order. The supplier receives the purchase order and checks if all goods are in stock. The order is then either confirmed or rejected. If the order was rejected the process ends, otherwise the supplier sends the invoice to the customer and ships the purchased goods. The process ends when the customer paid for his goods.

2.2 Business Process Management Lifecycle

Business processes pass different phases during their lifetime [9]. In general, the phases together form a lifecycle (see Figure 2.2) since a business process may pass all phases multiple times.

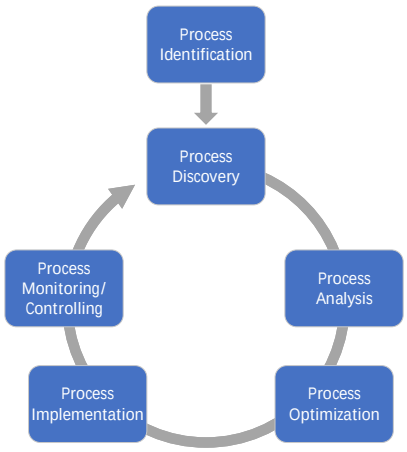


Figure 2.2: Business Process Management Lifecycle [9]

Initially, the business processes to be improved have to be identified. Furthermore, the mutual relations between the processes need to be determined. This phase is called *process identification*. Subsequently, the current state of each of the relevant business processes is documented by modelling the process as an "as-is" business process model in the *process discovery* phase. In the *process analysis* phase, issues associated with the "as-is" process are identified, documented and prioritized. Afterwards, in the *process optimization* phase, changes to the process which could help to solve the issues are

applied to the process model. In the *process implementation* phase, the optimized process is implemented. The final phase is the so-called *process monitoring and controlling* phase. Here, the redesigned process is executed and relevant data to measure its performance is collected and analyzed. During run time, new issues may arise that require the process to be further improved. Therefore, the process repeats the lifecycle's phases starting with the process discovery phase.

2.3 Knowledge-intensive Processes

This thesis uses Definition 2.3 describing *knowledge-intensive processes* (KiPs) [42].

Definition 2.3. ***Knowledge-intensive processes*** *(KiPs) are processes whose conduct and execution are heavily dependent on knowledge workers performing various interconnected knowledge-intensive decision-making tasks. KiPs are genuinely knowledge, information and data-centric and require substantial flexibility at design and run time.*

Standardized business processes mainly include routine work (e.g. production processes), which can be pre-specified in a process template by a domain expert. The activities of a standardized business process are foreseeable and, importantly, the specific order they are executed as well. This is why standardized business processes can be well supported by BPMSs, whereas KiPs still lack a comparable support.

KiPs represent a certain group of processes and differ from standardized business processes on the base of their characteristics [25] (cf. Figure 2.3). In particular, KiPs are *non-predictable*, *emergent*, *goal-oriented* and *knowledge-creating*.

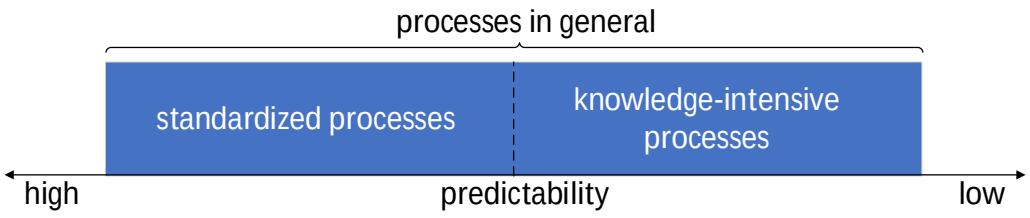


Figure 2.3: Correlation between standardized Business Processes and KiPs

To sucessfully achieve the process goal, knowledge workers typically divide the given goal into smaller, more manageable sub-goals (or milestones). To achieve their next milestones, knowledge workers then define the tasks to be performed next. The emergent nature of KiPs is evident in the fact that knowledge workers continously evaluate the current state of the process to derive new tasks and perform corresponding activities helping them to fulfill their (sub-)goal. A KiP usually involves multiple knowledge workers (potentially from different fields), who work together and communicate with each other to reach their common goal effectively. As knowledge workers gain more experience and expertise during the course of the process, KiPs are knowledge-creating.

In general, a high degree of uncertainty is involved in KiPs and, therefore, methodologies are used to better cope with the uncertainty and emergent nature of KiPs. However, methodologies are often specific for certain domains (e.g. the V model). In general, they can be abstracted by the generic Plan-Do-Study-Act (PDSA) cycle [25, 27] (cf. Figure 2.4). It has four phases: *planning work*, *performing work*, *studying work results* and *optimizing work plans*. Through the alternating planning and working phases, the KiP can be continuously adapted. In this way, knowledge workers may react to unforeseen events by taking them into account in the next planning phase.

Examples for KiPs are research projects, patient treatment processes in hospitals or criminal investigations.

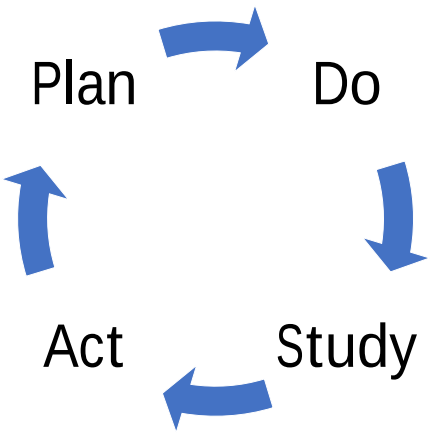


Figure 2.4: Plan-Do-Study-Act (PDSA) cycle

2.4 proCollab

2.4.1 proCollab Project

The proCollab[1] research project conducted at Ulm University [32] aims at establishing a holistic support for collaborative knowledge workers and their KiPs. Thereby, proCollab shall establish a generic, lightweight and lifecycle-based task management support for KiPs.
Basically, there are two different kinds of task lists:

- *to-do lists* as prospective task lists, which are used to plan future work, and
- *checklists* as retrospective task lists, which are used to evaluate work results.

Traditional paper-based task lists often cause *media disruptions* and only one person may access a task list at one time. Furthermore, if multiple knowledge workers edit the same task list, each having a task list on their own, then there will arise consistency and coordination problems. To solve these problems, proCollab uses digital task lists to support knowledge workers collaborating in a KiP. Involved knowledge workers may access their collaborative task lists at any time.

2.4.2 Knowledge-intensive Process Lifecycle

To realize a sustainable support for KiPs, the proCollab approach employs a lifecycle model. It consists of four phases: *orientation*, *template design*, *collaboration run time* and *records evaluation* (cf. Figure 2.5). Similar to the BPM lifecycle (cf. Section 2.2), the KiP lifecycle also aims at a continuous optimization and evolution of the underlying (knowledge-intensive) processes. This is achieved by improving the so-called *collaboration templates*. Knowledge workers may create a collaboration template in the template design phase. During the collaboration run time, so-called *collaboration instances* may be derived from the collaboration template. After the KiP has been completed, the deployed collaboration instances are analyzed and possible optimizations are applied to the collaboration template.

[1] ***pro**cess-aware Support for **Collab**orative Knowledge Workers*

The optimization and evolution of collaboration templates ensures that knowledge workers are better supported in their collaboration in KiPs, as they are provided with improved collaboration templates for future KiPs. In the following, the different phases are described.

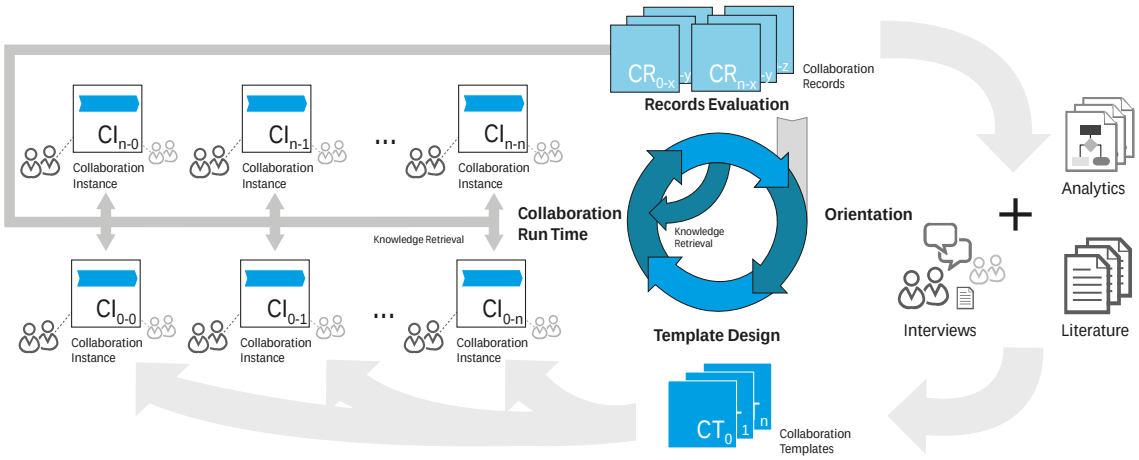


Figure 2.5: KiP Lifecycle Model [25]

Orientation Phase

The goal of the *orientation phase* is to capture the context of the collaboration and to identify the required resources (human and informational) as well as the type of collaboration. The result of this phase is based on interviews with the involved knowledge workers, literature on the subject and past collaborations of the same kind.

Template Design Phase

In the *template design phase*, the results of the previous phase are utilized to define *collaboration templates*. A collaboration template comprises tasks and features a common goal for the participating knowledge workers. Since collaboration templates should be reused for multiple collaborations, they should be designed generically so that they are suitable for similar collaborations.

Collaboration Run Time Phase

In the *collaboration run time phase*, a collaboration template is instantiated (*collaboration instance*) and, if necessary, adapted for the specific KiP (e.g. a project). Knowledge workers may use the information of past collaborations based on the same collaboration template – so-called *collaboration records* – while they execute their tasks. This way, they can additionally reuse the knowledge and experience stored in completed collaboration instances.

Records Evaluation Phase

Collaboration records are analyzed in the *records evaluation phase*. The goal of the analysis is to provide knowledge workers with better collaboration templates in the future. Therefore, collaboration templates should be optimized based on the collaboration records. For example, an optimized collaboration template should not contain any tasks that remained unused or were deleted in the majority of collaboration instances. Instead, an improved collaboration template should incorporate tasks that were added to the majority of collaboration instances at collaboration run time. The records evaluation phase represents a feedback loop to the collaboration run time phase, since it enables controlled evolution of existing templates for future collaborations.
The approaches for the automatic generation and evolution of task list templates presented and implemented in the course of this thesis form the basis for the records evaluation phase and thus for a continuous evolution of task list templates.

2.4.3 proCollab Entities

As a foundation and to realize the KiP lifecycle model (cf Section 2.4.2), proCollab employs *processes*, *task trees* and *tasks* as its key components (see Figure 2.6).

In practice, knowledge workers collaborate in the scope of *projects*, *cases*, or just *temporary endeavors* [25]. As these are all specific terms for KiPs, proCollab uses the term

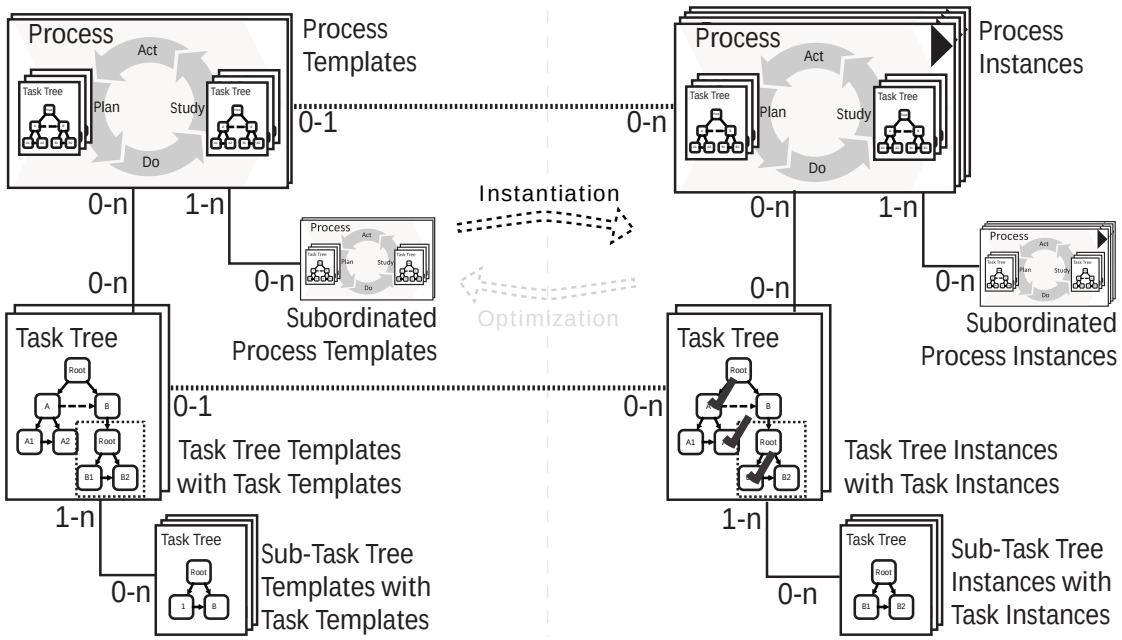


Figure 2.6: proCollab Entities [28]

process to generalize from these. A process can be arbitrarily nested and, therefore, may contain sub-processes. Each process features a goal the knowledge workers want to achieve through collaboration. In addition, each process is linked to *task trees* that allow knowledge workers to coordinate to successfully achieve the process goal.

A *task tree* is a generic data structure enabling the definition and usage of established task lists, i.e. to-do lists and checklists (cf. Figure 2.7). The latter are heavily used by knowledge workers to coordinate tasks among each other. In contrast to conventional tree structures, a recommended order is defined in which the tasks can be processed by knowledge workers. However, they can deviate from the given order at any time to adapt to the current situation of the KiP.

A particular *task* can contain an arbitrary number of child tasks that are supposed to be executed first in order to complete the actual task. Each task tree has a root node with a number of ordered child nodes (cf. Figure 2.7). These, in turn, may have other child nodes, which are also ordered. Apart from the root node, each node in the task tree is either a task or an embedded task tree. The root node is not shown as a task, but is needed to

create the tree data structure.

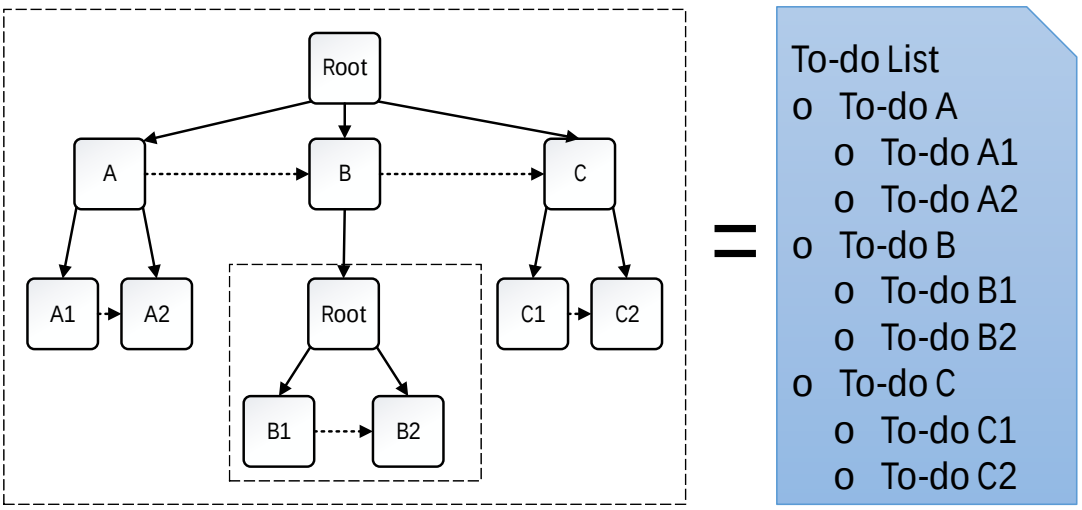


Figure 2.7: proCollab Task Tree Structure

A *task* stands for work to be done, it has a work description in the form of its name and its detailed description. To support both *to-do lists* and *checklists*, tasks may be specialized as *to-do* or *checklist items*.
In accordance with the KiP lifecycle, process, task tree and task templates enable knowledge workers to accelerate their planning and coordination by using and working with best practices and standards. For this purpose, a knowledge worker may select and instantiate a process or task tree template if required. Instantiation transforms a template into a corresponding instance which the knowledge workers can work with.
In the following the different templates and instances are introduced.

Process Template

A *process template* (PT) constitutes a process blueprint for a certain type of collaboration. It includes, among others, the goal of the collaboration, a relative deadline and, most importantly, corresponding task tree templates. Once a PT has been defined, knowledge

workers may instantiate it to create a process instance of a specific type, e.g. a project or a case.

Task Tree Template

A *task tree template* (TTT) contains a set of task templates and subordinated TTTs. It has a certain task tree type, e.g. *to-do list* or *checklist*. To-do lists enable prospective planning (i.e. tasks that have to be done), whereas checklists enable retrospective evaluations (i.e. to evaluate work results). TTTs contained in a PT are automatically instantiated as soon as the PT is instantiated. Alternatively, a TTT can be instantiated in the context of an existing process instance (PI).

Task Template

A *task template* is a blueprint for a task, it contains information about the task name, a detailed description, the task type (i.e. *to-do item* or *checklist item*) and the expected duration. A TT may be part of multiple TTTs with the result, that an update on a TT may automatically affect all linked TTTs. In addition, a task template may reference necessary resources (e.g. documents) required to complete the task.

Process Instance

A *process instance* (PI) refers to a certain process in use (e.g. a project or a case) and is either based on a PT or created as blank instance from scratch. Furthermore, it contains associated task tree instances (TTIs).

Task Tree Instance

A *task tree instance* (TTI) is created by either instantiating an existing TTT or by creating an individual one. It contains a set of task instances and subordinated TTIs as well as additional information about the creator and the PI it belongs to.

Task Instance

A *task instance* (TI) belongs to a (parent) TTI and is either instantiated from a TT or individually defined from scratch. It represents either prospective (to-do list item) or retrospective (checklist item) tasks. Knowledge workers may assign TIs to themselves or other participating knowledge workers to coordinate each other. TIs expose a state that can be changed by knowledge workers depending on the progress of the associated work task.

Records

For all instances, *records* are maintained accordingly. A record consists of a completed instance component and its change log. In particular, all changes applied to a TTI's task instances and subordinate task tree instances are logged in its change log.

2.4.4 proCollab Architecture

The proCollab approach was implemented in a sophisticated proof-of-concept prototype with a multi-layer architecture (cf. Figure 2.8). To provide knowledge workers high flexibility, proCollab may be used with web clients as well as mobile clients [12, 13, 17, 34, 40, 49]. To enable these clients to communicate with the server, various Representational State Transfer (REST [10]) interfaces are provided. The interfaces allow clients to search for task trees, retrieve them and also perform change operations.
The application logic layer represents the core of the prototype realizing the key services of the proCollab approach and its key components (e.g. processes, task lists, users). In addition, all change operations and state transitions are logged by the proCollab server in the change and execution logs of the corresponding instance.
The persistence layer stores proCollab entities in databases via the Java Persistence API (JPA) and the Java Content Repository (JCR).

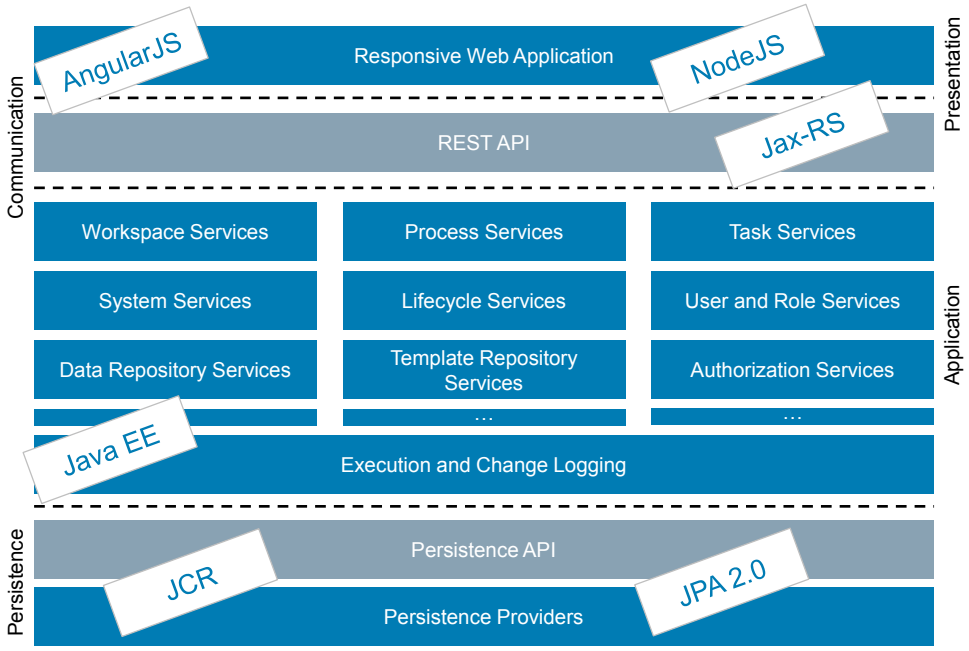


Figure 2.8: proCollab Prototype Architecture [32]

2.4.5 State Management

proCollab includes a generic and flexible state management concept supporting domain-specific states [28]. The proCollab state management mainly relies on *reference state models*, *state models* and *state model instances* (cf. Figure 2.9). The set of stateful key components includes PTs, PIs, TTTs, TTIs, TTs and TIs.
Reference state models define the states and state transitions that are granted for all entities of a certain type (e.g. all PTs). In turn, state models may refine specific states of a reference state model to meet domain-specific requirements. Finally, each proCollab key component references a state model instance that interprets a state model and features a current state that is changeable.

Reference State Models

A *reference state model* consists of a *state transition graph*, a set of refinable states and a scope. In general, a state transition graph is composed of states and state transitions.

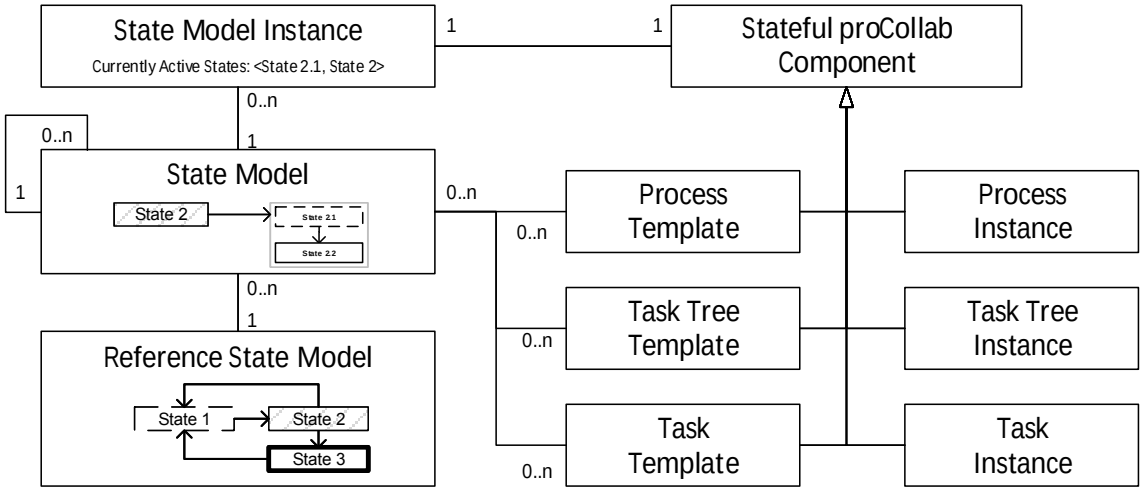


Figure 2.9: proCollab State Management [28]

Furthermore, a start state and one or more final states are defined by the state transition graph. The scope of a reference state model determines for which stateful proCollab entities the reference state model can be used. For example, a reference state model with the scope "task tree instance" is used as the basis for all state models of task tree instances.

State Models

State models are linked to a reference state model and adopt the scope of the associated reference state model. Furthermore, state models are used to refine reference state models and to incorporate domain-specific states. The latter may constitute phases of methodologies (e.g. Scrum) employed by the involved knowledge workers. Figure 2.10 shows the refinement of the state *In Progress* to represent the individual phases of a Scrum sprint (*initiate*, *plan and estimate*, *implement*, *review and retrospect*, *release*) [38].

State Model Instances

State model instances make proCollab components stateful. Every state model instance references a state model and contains a sequence of currently active states. When a

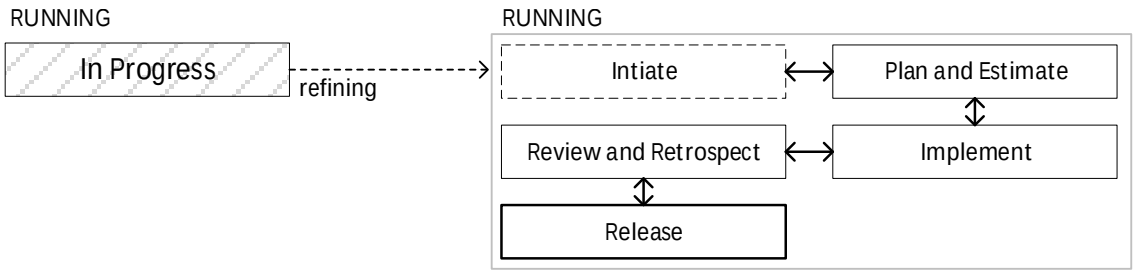


Figure 2.10: Phases of a Scrum Sprint represented as Refined States [28]

stateful proCollab component is created, a corresponding state model has to be selected. Based on the latter, a state model instance is derived and linked to the stateful proCollab component. Each proCollab template component may reference a state model supposed to be used for the derived instances. For example, a process template can link to a specific state model with the scope "process instance", which should be used to create a state model instance for a derived process instance.

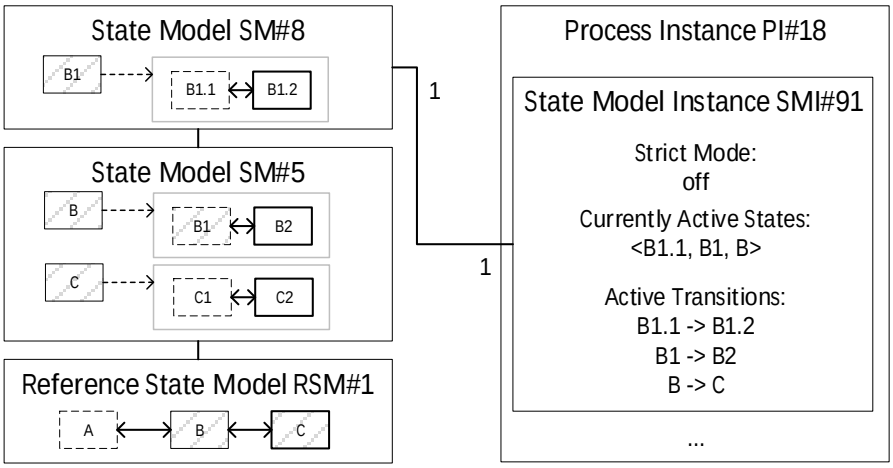


Figure 2.11: Exemplary State Model Instance with Corresponding State Model [28]

The currently active states of a state model instance are arranged from the most specific state to the most abstract one, i.e. refining states are placed before the state being refined. (cf. Figure 2.11). If the *strict* constraint is set, all refining states must be finished before the superior state can be completed and an outgoing state transition can be triggered to switch to another active state.

2.4.6 Specialization Types

To fully exploit the potential of the proCollab state management concept, proCollab uses *specialization types* that allow to enhance the generic data structures of processes, task trees and tasks (cf. Figure 2.12). Accordingly, the most common specialization types are *process types*, *task tree types* and *task types*. proCollab offers six pre-defined specialization types: the process types *project* and *case* for process templates and instances, the task tree types *to-do list* and *checklist* for task tree templates and instances and the task types *to-do item* and *checklist item* for task templates and instances. Every specialization type may refer to a set of state models that can be used for templates and instances (depending on the scope of the reference state model). As a result, the process of selecting an appropriate state model is facilitated significantly and knowledge workers are enabled to more effectively create customized process, task tree and task instances.

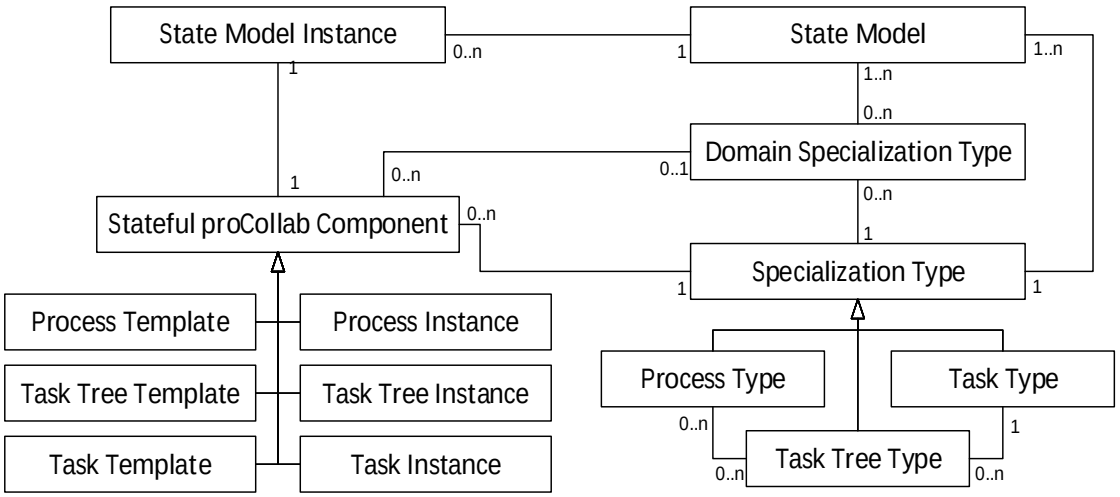


Figure 2.12: proCollab Specialization Entities [28]

Furthermore, every specialization type exposes a *temporal perspective* to support knowledge workers according to the PDSA cycle (cf. Section 2.3). The temporal perspective denotes whether proCollab components may be used for planning (*prospective* temporal perspective), for quality assurance (*retrospective* temporal perspective) or for both aspects (*hybrid* temporal perspective). For example, the *to-do list* task tree type and the *to-do item* task type both expose the *prospective* temporal perspective, whereas the *checklist* and

checklist item types expose the *retrospective* one. Specialization types may be interlinked to ensure that certain types are always used together in combination (cf. Figure 2.12). For example, the *checklist* task tree type is interlinked with the *checklist item* task type to ensure that a checklist only contains tasks of type *checklist item* (and none of type *to-do item*).

2.4.7 Object-specific Role-based Access Control

proCollab provides a powerful role and permission model for access control, based on the concept of object-specific role-based access control (ORAC) [26]. ORAC enables the close integration of access control with the given object model as well as the support of roles in a fine-grained, object-specific way. ORAC comprises the key components *guarded objects*, *privileges*, *object aware roles*, *organizational entities*, *agents* and, importantly, *object-aware role assignments* (cf. Figure 2.13). *Guarded objects* are objects protected by ORAC, i.e. an agent can only manipulate data if he was granted access to this action by ORAC. *Organizational entities* allow to model the organizational context of agents and consist of organizational units (e.g. *HR department*), organizational roles (e.g. *director*) and abilities (e.g. *office skills*). The organizational units may be nested to constitute the typical structure of an organization. Organizational roles may be used to indirectly assign *privileges* to users having the respective role.

Object-aware role assignments are used to tie together the different components comprising ORAC, namely agents, object-aware roles and guarded objects. Each object-aware role contains a key scope and optionally a number of additional scopes. These scopes are used to assign privileges regarding different object types to the object-aware roles and allow the creation of object-aware role assignments. These privileges determine which actions can be performed on a guarded object and in which context the privileges are applicable. To allow for a hierarchical application of privileges, an object-aware role may reference a set of hierarchical privileges for every scope. Furthermore, a set of entity-related privileges may be referenced to provide a rich modeling of object-aware roles.

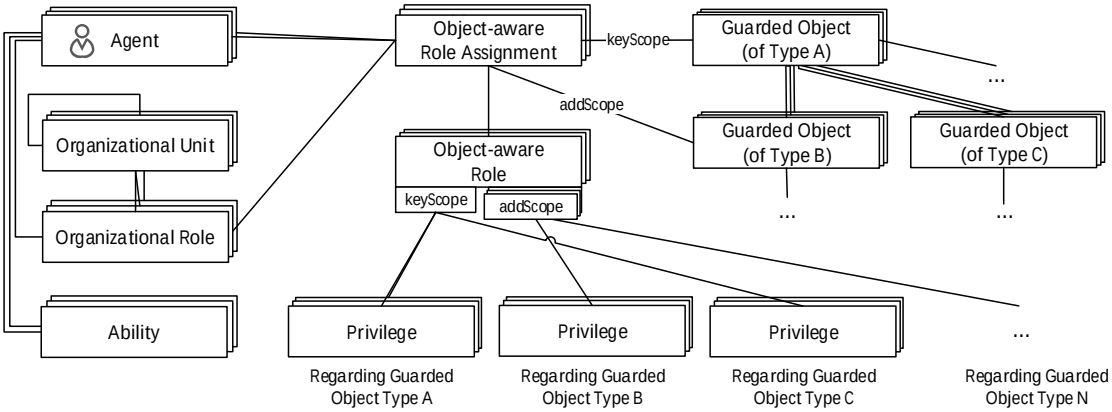


Figure 2.13: proCollab Object-Specific Role-Based Access Control [26]

2.4.8 Comparison of the proCollab Prototypes

In the course of the proCollab project, two prototypes have been developed. Based on the approach of rapid prototyping, a first prototype was developed to evaluate the concepts in terms of technical feasibility and conceptual appropriateness. Based on the lessons learned, a second prototype, which includes a new server application and new web client, has been developed for the recent years. This section explains the differences between the two proCollab prototypes and highlights the implications to this thesis.

Entities

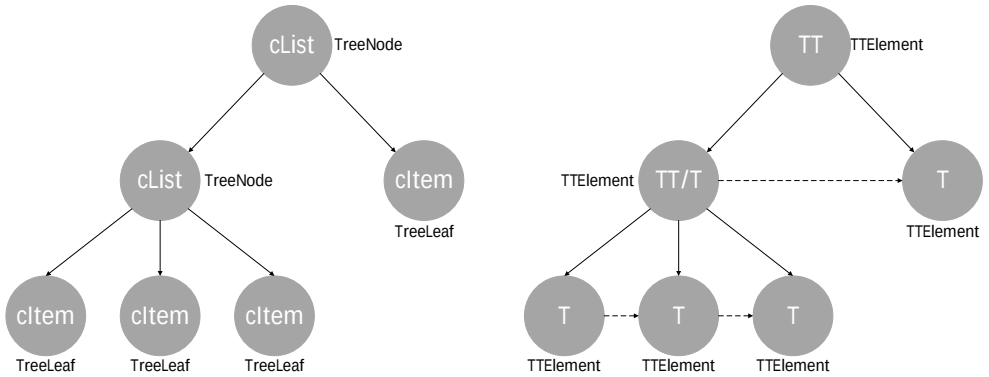
Table 2.1 provides a mapping of the entities of the first prototype with the corresponding entities of the second prototype. The main components of proCollab were already present in the first prototype, but their design has changed considerably. In detail, entities for processes, task lists and tasks exist in both prototypes. In the first prototype there was no possibility to specialize the generic entities. For example, a cList had no specialization type. In the second prototype, on the other hand, the main components can be specialized using specialization types (cf. Section 2.4.6). Accordingly, the most common specialization types are process types, task tree types and task types. For example, a task tree can be specialized by a task tree type as to-do list or checklist. Consequently, a task is specialized as a to-do item or checklist item.

Table 2.1: Entities of the old Prototype with the corresponding Entities of the new Prototype

old entity	new entity
cFrameType	process template
cFrameInstance	process instance
cListType	task tree template
cListInstance	task tree instance
cItemType	task template
cItemInstance	task instance

Structure of the Task Lists

The general structure of the task lists in the form of a tree structure is the same in both prototypes. However, the nesting and, in particular, the implementation of the tree structure has changed. Figure 2.14 shows the two different approaches for representing the tree structure. Figure 2.14 (a) represents the concept of the tree structure in the first prototype. An important distinction was made between tree nodes and tree leaves. A node could have child nodes or subordinate leaves. All nodes were realized as cLists. In turn, there were no cItems with subordinate nodes. Figure 2.14 (b) shows the concept of the tree structure (task tree) in the second prototype. It allows any nesting of tasks and subordinate task trees. In contrast to the first prototype, it is possible that a simple task has subordinate elements in the form of a task tree or tasks, i.e. the second prototype is not limited to the fact that only task trees can have child elements.



(a) 1. Prototype (b) 2. Prototype

Figure 2.14: Comparison of Task List Structures in the proCollab Prototypes

In the first prototype, the entities were explicitly implemented as nodes and leaves of a tree by the two interfaces *TreeNode* and *TreeLeaf*. Child elements were directly added to the corresponding TreeNode. In contrast, both task trees and tasks of the second prototype implement the common interface *TaskTreeElement*. Thus task trees and tasks may be nested arbitrarily. The tree structure in the second prototype is created using a map with adjacency lists. If a task tree element has child elements, it has an entry in the map that refers to a list of the corresponding child nodes (cf. dotted lines in Figure 2.14 (b)), which in turn may have their own subordinate elements.

State Management

The state management of the stateful entities strongly differs in the proCollab prototypes. In the first prototype, the states were pre-defined in the source code for each entity. In comparison, the second prototype provides a sophisticated state management with reference state models, state models and state model instances (cf. Section 2.4.5). As a highlight, the new concept allows changes of the given states and state models at run time, while in the first prototype the code of the respective entity had to be adapted. Furthermore, in contrast to the second prototype, it was not possible to refine states in the first prototype and thus to meet domain-specific requirements.

Access Control and User Roles

Both proCollab prototypes realize a form of access control based on various user roles. The first proCollab prototype employs a simple role set consisting of five different roles: administrator, manager, employee, user and no role. To access task lists, the user must be logged in, have at least the role of a user and be involved in the corresponding process to which the task list belongs. Any method involving modifying, inserting or deleting may only be executed if the logged in user has at least the role of a manager of the corresponding process. Access control is performed by checking whether the logged in user has the necessary role to execute the corresponding method.
In contrast, the second proCollab prototype employs an advanced role system and conse-

quently a more complex access control. There are 15 pre-defined user roles, however, additional roles may be added as required, i.e. the role system is extendable at run time. Furthermore, a user may have several roles at a time. Access control in the second prototype is two-fold. For each method to be checked by the access control, the rights required to execute the method are defined separately. To check whether a user is allowed to execute a certain method, it is checked whether the user has the required authorization through one of his roles. Since the required authorizations are set individually for each method, this procedure allows a more detailed access control.

2.5 Task Tree Change Operations

As the planning and working phases of a KiP typically alternate, it is important that task lists can be edited allowing them to be adaptable to new requirements. As a result, knowledge workers intensively edit task lists, especially to-do lists, at run time to adapt to current events. Therefore, selected, well-defined change operations for editing task lists are required. Changes applied to a task list are recorded in its change log and serve as the starting point for future analysis and optimization.
Task tree change operations are either of basic character (atomic) or of a more complex one [47]. However, complex change operations may typically be represented as a series of simple changes.
Considering task trees, there are six basic change operations:

- *insert* a task or task tree at a given position,
- *delete* a certain task or task tree, and
- *update* a certain task (e.g. change its priority).

Both the insert and delete operations change the structure of the task tree, whereas the update operation does not.
To allow knowledge workers to edit task lists more conveniently, a set of more complex high-level change operations on task trees should be provided. Such relevant high-level change operations applied to task trees are:

- *move* a task or task tree to a different position,
- *replace* a task or task tree with another task or task tree, and
- *swap* the positions of two tasks or task trees.

To demonstrate the mapping of a complex, high-level change operation as a sequence of atomic operations, Figure 2.15 (a) shows the high-level *move* operation and Figure 2.15 (b) shows the corresponding atomic operations leading to the same result. The change operation on the left side directly applies the move operation to "*Task C*" and moves it to position 1 right below "*Task A*". On the right side, two change operations are applied: first "*Task C*" is inserted below "*Task A*" and then "*Task C*" is deleted at the old position.

Change operations made during run time represent adjustments of the task list instances. By adapting task list instances, new knowledge that is necessary for successfully completing the KiP is incorporated into them. An analysis of the completed task list instances and thus the applied change operations may reveal improvements for the task list template on which the task list instances are based. Therefore, change operations are an important part of the task list lifecycle, as they are the starting point for further analysis.

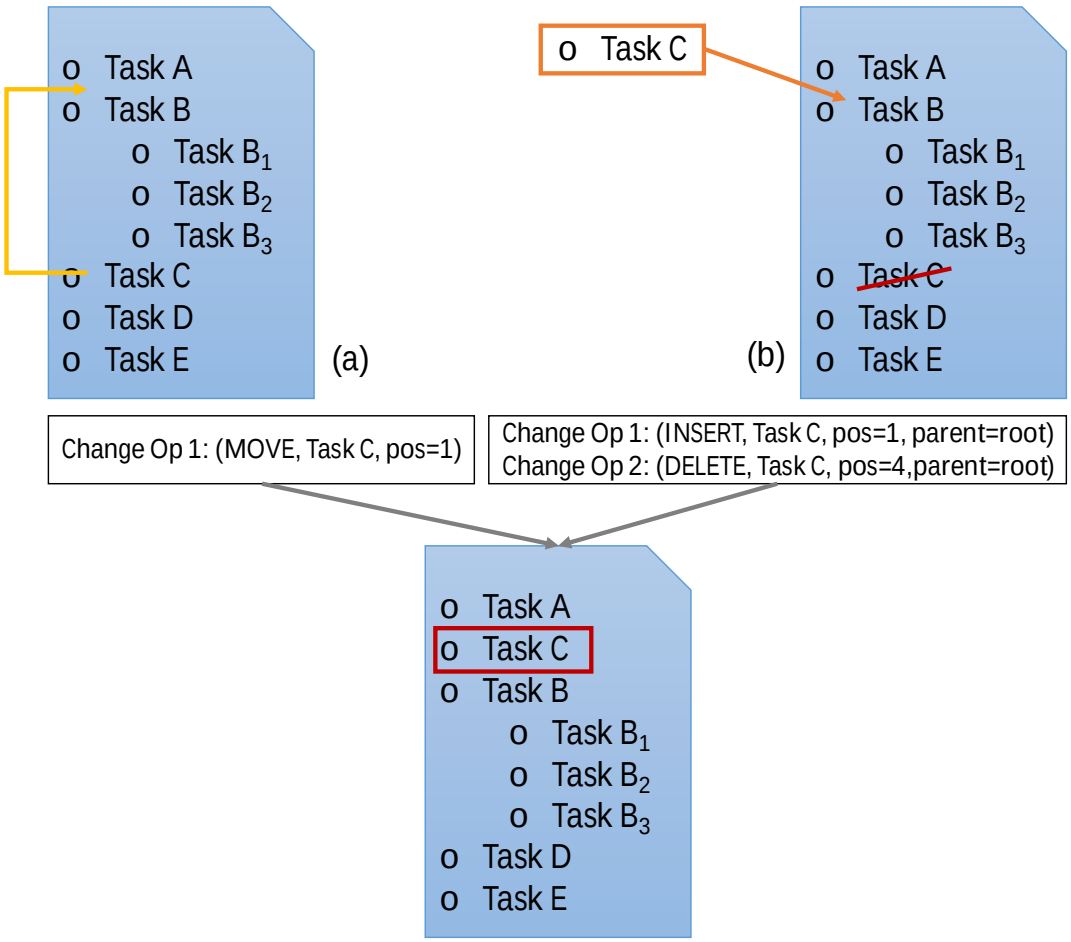


Figure 2.15: Task List Example with Change Log

2.6 Process Mining

In the course of this thesis, *process mining* is used to analyze change operations applied to task lists at run time and to identify their change process. The change process forms the basis for the optimization of the underlying task list template, since possible optimizations may be derived from it. By analyzing the change operations and the resulting optimization of task list templates, the optimization phase of the task list lifecycle may be realized.

Process mining algorithms are used to extract knowledge from logs and thus serve as a basis for process analysis [44]. Modern information systems, including BPMSs, create log files at run time that may serve as a foundation to apply process mining algorithms. A typical event log of a BPMS consists of a reference to the process instance, the name of the executed activity, its state and a timestamp. Furthermore, it is logged who performed the activity. By calculating the difference between an activity's begin and end timestamp its duration can be determined. The order of executed activities can be easily reconstructed with the help of an event log by grouping the events for every process instance and sorting them by their timestamps.
Process discovery (cf. Section 2.2) is a characteristic goal of process mining. Its purpose is to mine a process model from a given event log. It is not always easy to define a process model which is why this technique is useful to automatically derive a process model from the given event log. Another characteristic goal is *conformance checking* detecting deviance between an existing process model (*"to-be"*) and the recorded behavior in the event log (*"as-is"*). Furthermore, there are several other aspects that can be analyzed by applying process mining techniques, for instance, identifying bottlenecks in a process.

2.6.1 Change Mining

Change mining uses process mining algorithms to identify the change process of process instances by analyzing their change logs [14]. Change logs are similar to execution logs, but instead of the execution order of activities the change operations applied to the process instance are logged. Change logs consist of the change operation type (for instance insert, delete, update), the changed activity, the performing user, a timestamp and, optionally, the

position of the affected activity in the sequence of activities within the process instance. The latter is optional, since some operations (e.g. update) do not necessarily require this. The order of change operations can be reconstructed by ordering the change operations according to their timestamps.
With change mining, the change process for process instances is retrieved using their change logs (cf. *process discovery*). The change process illustrates the applied change operations, their order and their frequency. By analyzing the change process, different change variants are discovered. When considering the frequency of change operations, it is possible to find sequences of frequently applied change operations. The most frequent change operations may then be applied to the given process model to optimize it. Instead of applying the most frequent change operations to the original process model, it is possible to create a copy and apply the changes to it, so that an additional process model is generated (i.e. a variant of the given process model).

2.6.2 Causal Net

Process mining results can be represented in a variety of technical, process-based notations, such as classic Petri nets [30], Workflow nets [43], Event-driven Process Chains (EPCs) [36] or BPMN [29]. A particularly interesting notation to illustrate observed behavior in an analyzed log are *causal nets* (C-nets) [45, 44]. The latter were specifically designed for the needs of process mining techniques. A C-net is a graph structure in which each node represents an activity connected by edges. Figure 2.16 shows an example C-net. At the beginning, only the start node is activated and directly succeeded by A. A is then executed and afterwards one of A's four different output bindings is selected – just B, just C, just D, or B and C.

Each node has sets of potential *input* and *output bindings* (indicated by the black dots in Figure 2.16). Dots on the outgoing edges of a node represent the output bindings, dots on the incoming edges represent the input bindings. Bindings with multiple nodes are connected through an additional arc (cf. output binding *B and C* in Figure 2.16). The start node has no input binding, whereas the end node has no output binding. Each binding represents an alternative path. A node is activated as soon as one of its input bindings is

fulfilled, i.e. all of the bound nodes have been executed successfully before. After a node was executed, an output binding is chosen and all of its bound nodes have to be visited before executing the end node.

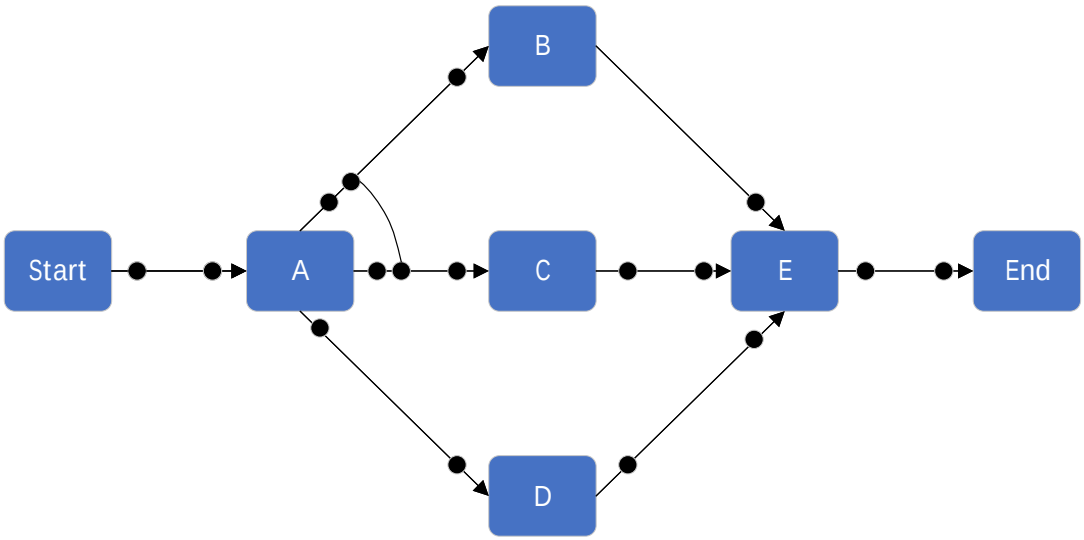


Figure 2.16: Example C-net

The representation of a change process in the form of a C-net is particularly useful since the C-net can be easily analyzed. The different change process variants, i.e. paths through the C-net, can be identified by traversing the C-net and taking each output binding into account. In addition, C-nets are widely used and many process mining algorithms (e.g. the *multi-phase mining algorithm* (cf. Section 6.1.3)) are based on C-nets [6, 7, 8, 45].

3

Task List Lifecycle Management

The proCollab task list lifecycle management consists of two different components: *task list template generation* and *task list template evolution*. Task list template generation enables knowledge workers to generate a task list template out of existing task list instances. As a result, the knowledge and experience contained in completed task list instances is utilized to be combined in a new task list template that may be used in future KiPs. Task list template evolution is used to optimize an existing task list template. First, change operations, which were applied to task list instances derived from the task list template, are analyzed to identify the most frequent changes. In a second step, the most frequent changes are applied to the task list template. This approach reuses the knowledge contained in the different task list instances to optimize an existing task list template.

Figure 3.1 illustrates the concept of the proCollab task list lifecycle management (cf. Section 2.4.2). At the beginning, there is the orientation phase in which important information about the KiPs is collected. During the collaboration run time phase, different task list instances are created in case there is no suitable task list template. With the help of the newly created task list instances, the KiP is successfully completed. Afterwards, the task list template generation generates a common task list template from the completed task list instances.

If a similar KiP occurs in future, the previously generated task list template serves as a starting point for the knowledge workers. The corresponding task list template is then instantiated during the collaboration run time phase and supports the knowledge workers collaborating in the KiP. Changes made to the task list instances by knowledge workers at run time are analyzed in the task list template evolution phase after the completion of several similar KiPs. The most frequent changes are identified and then applied to

the original task list template or one or several new, specialized task list templates are introduced.

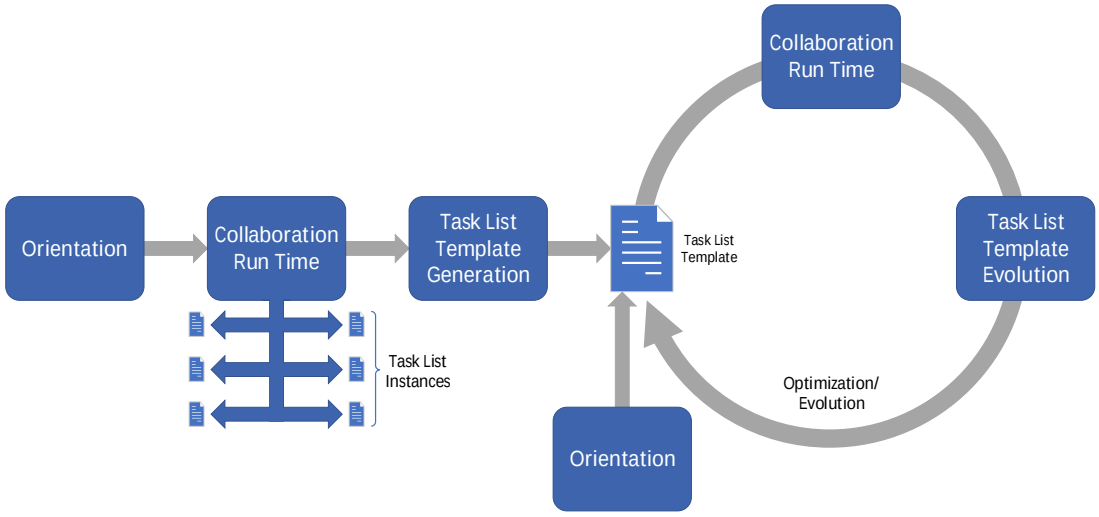


Figure 3.1: Task List Lifecycle Management Overview

To emphasize the challenges and difficulties of task list lifecycle management, the problem statement is presented in Section 3.1. Section 3.2 briefly explains the task list template generation approach followed by a brief explanation of the task list template evolution approach in Section 3.3. Both approaches will be explained in detail later in the corresponding Chapters 5 and 6. In Section 3.4, the implementation and architecture behind task list lifecycle management is discussed.

3.1 Problem Statement

Knowledge workers mainly rely on task lists for coordination and collaboration in a KiP. The most common task lists are paper-based and therefore cause major problems. The use of paper-based task lists significantly hinders the synchronization and coordination of knowledge workers in a KiP, since only one knowledge worker may access a task list at a time. In addition, paper-based task lists cause media disruptions. Although it is possible to create task list templates with paper-based task lists, these task list templates

are not supported by a lifecycle and are therefore not suitable for a holistic support of knowledge workers in a KiP. Digital task lists, on the other hand, solve the main problems of paper-based task lists (*synchronization*, *availability*), but are not supported by a lifecycle. proCollab aims at supporting knowledge workers throughout all phases of the KiP lifecycle by using digital task lists. However, at the moment, proCollab offers the possibility to create task list and process templates, which may be defined during the template design phase by knowledge workers to enable the reuse of existing knowledge in the form of best practices and standards. These templates may be instantiated by knowledge workers at run time. Unfortunately, it is not always possible to predefine a task list template, because the characteristics of a KiP (*uncertainty*, *emergence*) make it rather difficult to define a suitable template in advance. Alternatively, a task list template may be automatically derived from already existing, comparable task list instances, which have been completed by knowledge workers. Since completed task list instances have already been successfully executed, the task list template derived from them is a promising candidate for similar KiPs in the future. Through this approach, the *knowledge-creating* nature of KiPs is exploited to generate a task list template based on the knowledge of the completed task list instances. During the collaboration run time phase, a task list template is typically instantiated by knowledge workers in the shape of task list instances in different PIs. At run time, knowledge workers may perform change operations on task list instances for various reasons. For example, a task list template may be *too generic*, i.e. more specific tasks or various sub-tasks that are required at run time are missing. In addition, the task list template may contain tasks that are *not required* at run time and be removed therefore. Tasks can also be *arranged incorrectly*, i.e. the ordering does not correspond to the desired execution sequence of the knowledge workers.

However, these problems are very difficult to prevent, as the characteristics of a KiP (*uncertainty*, *emergence*) hamper the definition of a suitable task list template beforehand. Nevertheless, if knowledge workers have to perform redundant tasks by always adapting a task list instance in a similar way shortly after instantiation, their workload can be significantly reduced by the continous and automatic incorporation of the most frequent changes into the existing task list template. Unfortunately, proCollab currently lacks an

automatic evolution phase, in which existing task list templates are optimized based on the derived task list instances (cf. Figure 3.2).

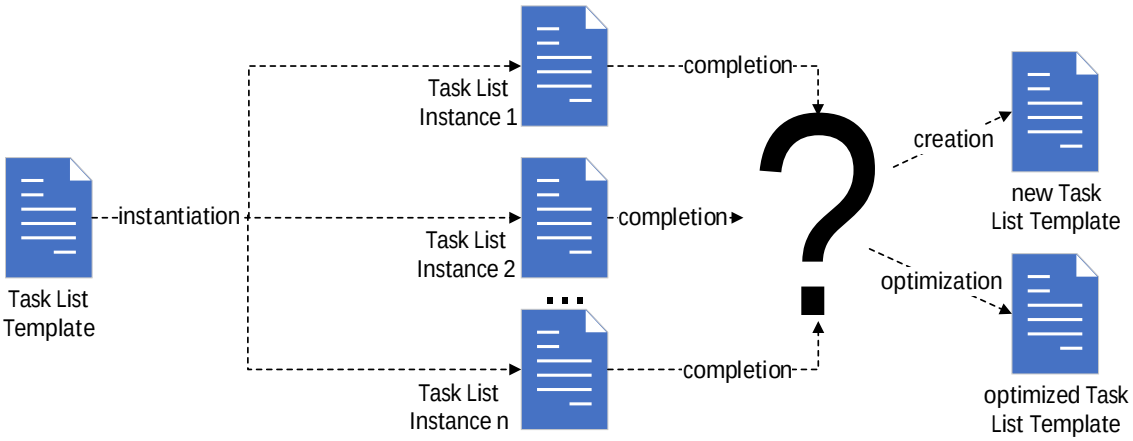


Figure 3.2: Idea of reusing Knowledge

3.2 Task List Template Generation

The *task list template generation* aims to generate a task list template from different but comparable task list instances. It is utilized when no suitable task list template exists but several similar task list instances have already been successfully completed by knowledge workers in the course of comparable process instances. At run time, knowledge workers build the PIs and the included task list instances from scratch to support the corresponding KiP. Since KiPs are knowledge-creating, the knowledge gained from the KiP is incorporated into the respective process instance and its task list instances. After multiple comparable process instances have been successfully completed, they contain task list instances with task instances required to successfully complete the KiP. It is possible to generate a common task list template using the task list instances from the comparable process instances. This proceeding is illustrated in Figure 3.3. At first, there is no suitable process template for the corresponding KiP. Therefore, the knowledge workers build up process and task list instances from scratch in order to support their collaboration in a KiP. When enough task list instances have been completed for similar process instances, the task list template generation may be used to generate a common, suitable task list template for

the completed task list instances. The generated task list template may be used for future occurrences of similar KiPs.

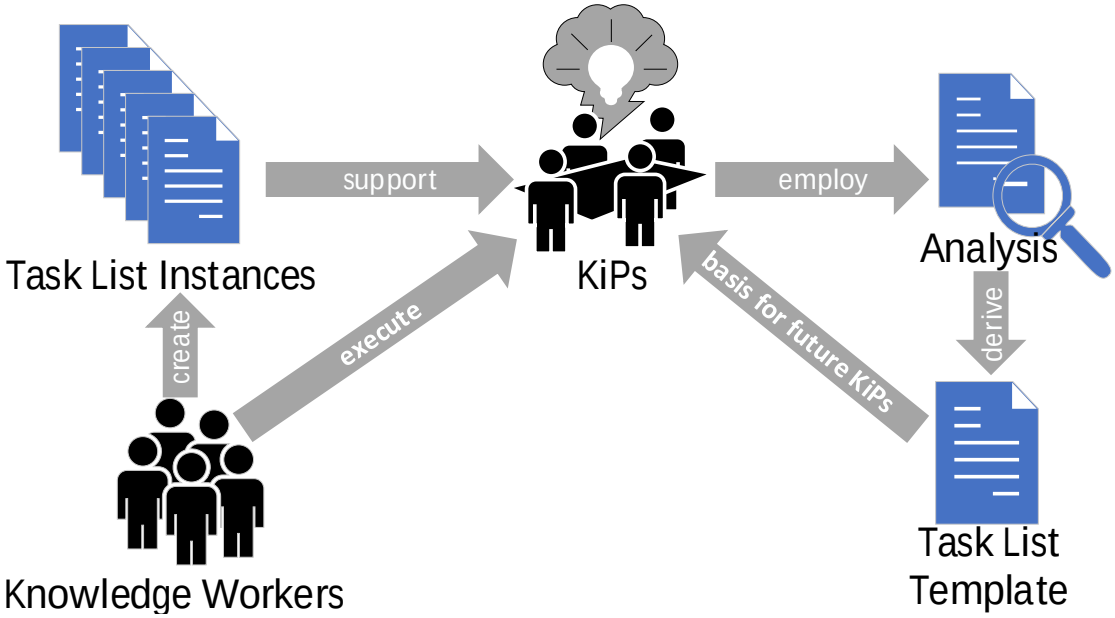


Figure 3.3: Task List Template Generation

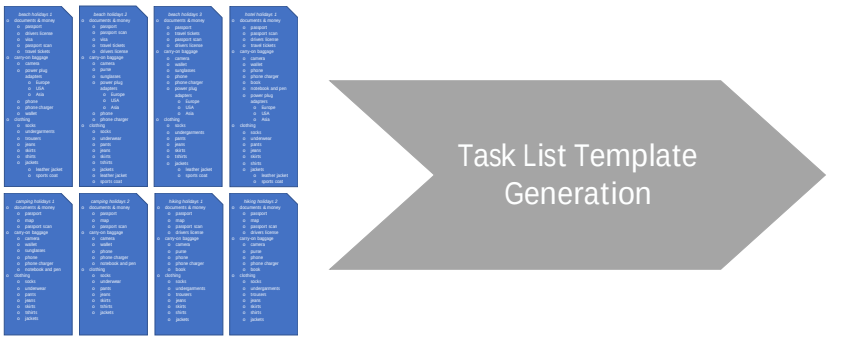
Illustrative Example

To demonstrate how the task list template generation works, an illustrative example is given now: eight knowledge workers would like to plan their holiday trips. For this purpose, each of them creates a separate task list instance in the shape of a checklist. Three knowledge workers are planning a holiday on the beach, one wants to go on a city trip, two are planning a hiking holiday and two others would like to camp. The different task list instances are shown in Figure 3.4.

Although the knowledge workers plan different trips, the checklists share a similar structure and contain similar tasks. As some tasks occur in all task list instances having a specific relation to other tasks, certain conditions can be identified for the arrangement of tasks that apply in all task list instances and, therefore, also have to apply in the task list template to be generated. After the knowledge workers have all successfully completed their journeys, they want to benefit from the gained experience and derive a common task list template for

future journeys. The eight task list instances they created are analyzed using the task list template generation pipeline. First, the similarity analysis is performed and similar tasks from the various task list instances are grouped into similarity groups (cf. Section 4.4). The order matrices (cf. Section 5.2.1) and the aggregated order matrix (cf. Section 5.2.2) are then created and non-frequent tasks are sorted out. The cluster mining algorithm (cf. Section 5.2.3) then clusters the tasks and builds the common task list template. Figure 3.5 shows the resulting task list template on the right. It combines the most common checklist items from the eight task list instances and provides a framework for future journeys. The task list template is generic as it contains the common checklist items from the four different trip types. The task list instances derived from the generated task list template may then be adapted accordingly for each trip type, for example by adding trip-specific items.

Figure 3.4: Example Task List Instances



Generated TTT
- documents & money
  - map
  - passport
  - travel tickets
  - drivers license
- carry-on baggage
  - camera
  - wallet
  - power plug adapters
    - Europe
    - USA
    - Asia
  - phone
  - phone charger
- clothing
  - socks
  - undergarments
  - underwear
  - pants
  - jeans
  - skirts
  - shirts
  - tshirts
  - jackets
    - leather jacket
    - sports coat

Figure 3.5: Resulting Task List Template derived from given Task List Instances

3.3 Task List Template Evolution

The purpose of the *task list template evolution* is to optimize an existing task list template by analyzing the change operations performed on the task list instances at run time and then applying the most frequent changes to the task list template. To provide knowledge workers with the most suitable templates at all times, it is necessary that task list templates can be evolved continously. By applying the most frequent change operations to the corresponding task list template, the knowledge and experience gained by knowledge workers during their collaboration in the KiP are extracted and included in the task list template so that it is available for future KiPs. Otherwise, the knowledge and experience would remain unnoticed in the completed instances and would not be reused instead.

Knowledge workers instantiate a suitable task list template at run time and coordinate themselves in a KiP using the derived task list instances (cf. Figure 3.6). They may apply changes to the relevant task list instance at run time if it does not meet their requirements. Supported change operations on task list instances are adding, deleting and updating task instances and task list instances (cf. Section 2.5).
Once multiple task list instances have been successfully completed for similar KiPs, the various change operations can be analyzed using the task list template evolution pipeline. The most frequent changes are identified and change variants are created. The knowledge workers may then select one or several change variants that are to be used to optimize the existing task list template.
An existing task list template can either be replaced, updated or additional task list templates can be added. If the task list template is to be replaced, the original task list template will be deleted and one or more new task list templates will be added. If the task list template is updated, the changes will be applied to the original task list template. If one or more new task list templates are to be added, the original task list template will not be changed, but a new, specialized task list template will be created.

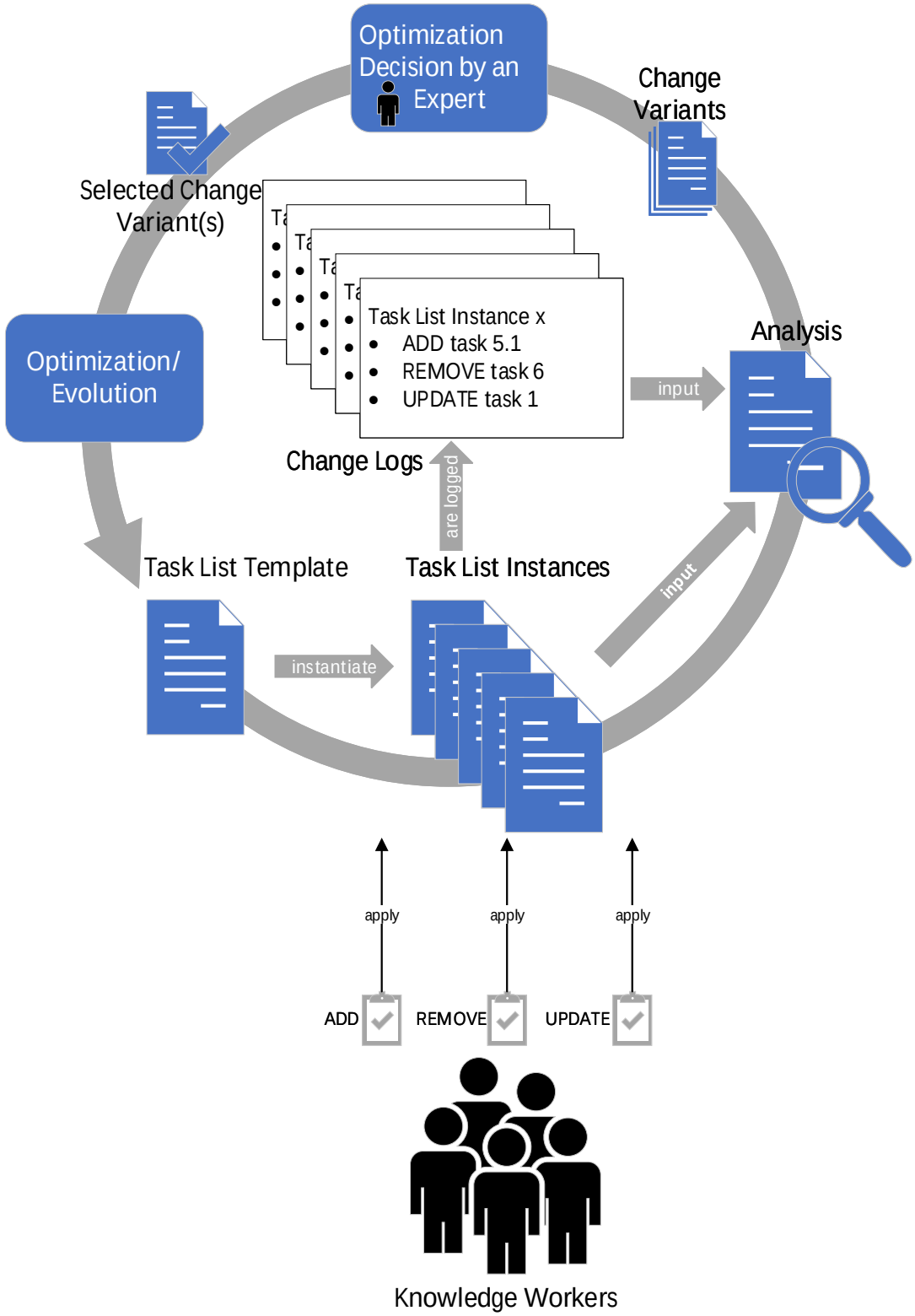


Figure 3.6: Task List Template Evolution

Illustrative Example

For a better understanding of how the task list template evolution works, this approach is also explained using an illustrative example: ten knowledge workers want to plan a holiday using an existing travel list template. For this purpose, everyone instantiates the task list template and edits the corresponding task list instance. Since the knowledge workers have different expectations of the ideal holiday, they plan their holidays in various ways. They apply three different variants of changes to the task list instances, as it can be seen in Figure 3.7: three knowledge workers apply the changes from the set of change operations *Change Ops 1*, five knowledge workers perform the changes from *Change Ops 2* and two knowledge workers perform the changes from *Change Ops 3*. The applied changes are recorded in the change log of the respective task list instance.

After the knowledge workers have finished planning their trips, they want to optimize the existing task list template based on the task list instances they have customized. For this purpose, the applied change operations are analyzed.

First, similarity analysis (cf. Section 4.4) is used to identify similar change operations by analyzing the change logs of the task list instances. Afterwards, a causal net (cf. Section 2.6.2) is created using the multi-phase mining algorithm (cf. Section 6.1.3). The C-net is then analyzed and change variants are derived from it. The knowledge workers then make an optimization decision by selecting one or more change variants to update or replace the existing task list template or to add additional specialized task list templates. Figure 3.8 shows possible change variants. As an assumption, the knowledge workers choose *Change Variant 5* to update the existing task list template. The resulting optimized task list template is shown in Figure 3.9.

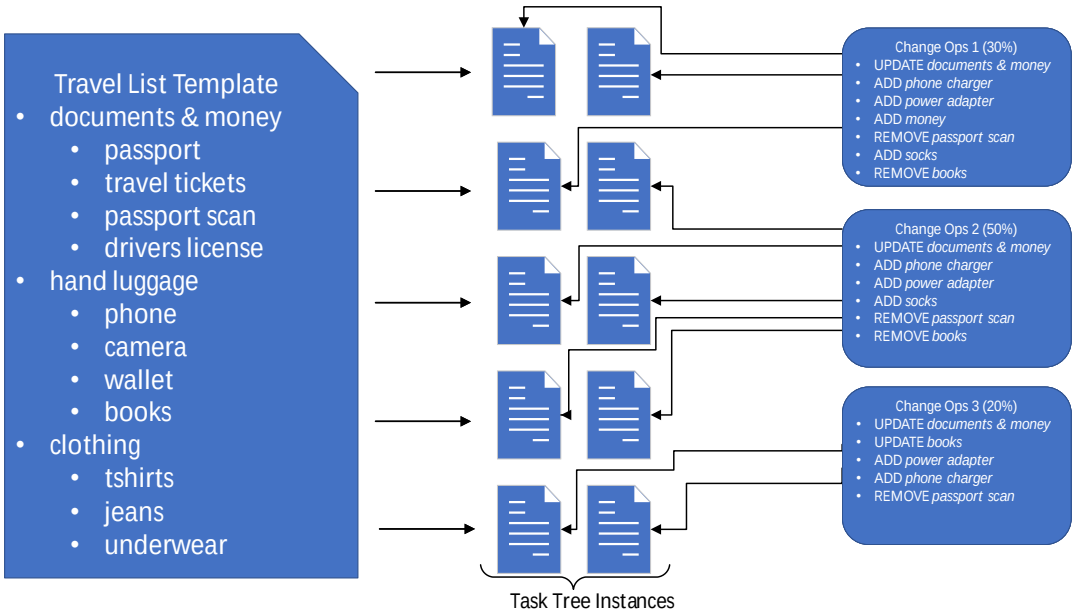


Figure 3.7: Exemplary Application of various Change Operations to Task List Instances

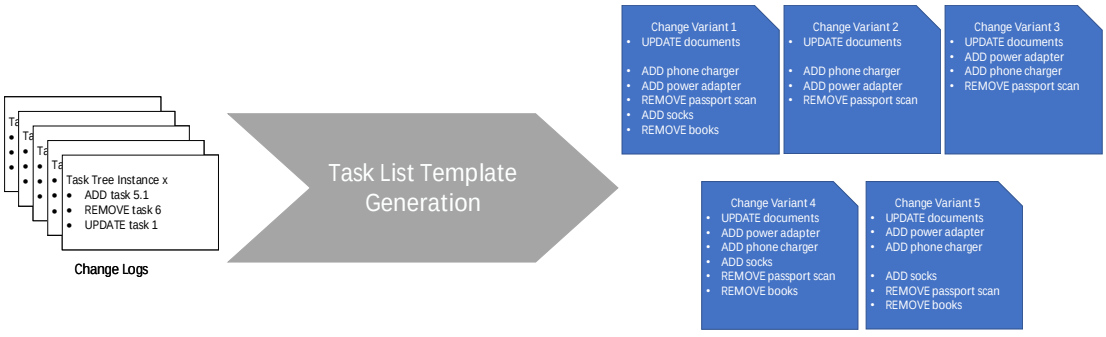


Figure 3.8: Found Change Variants based on Change Logs

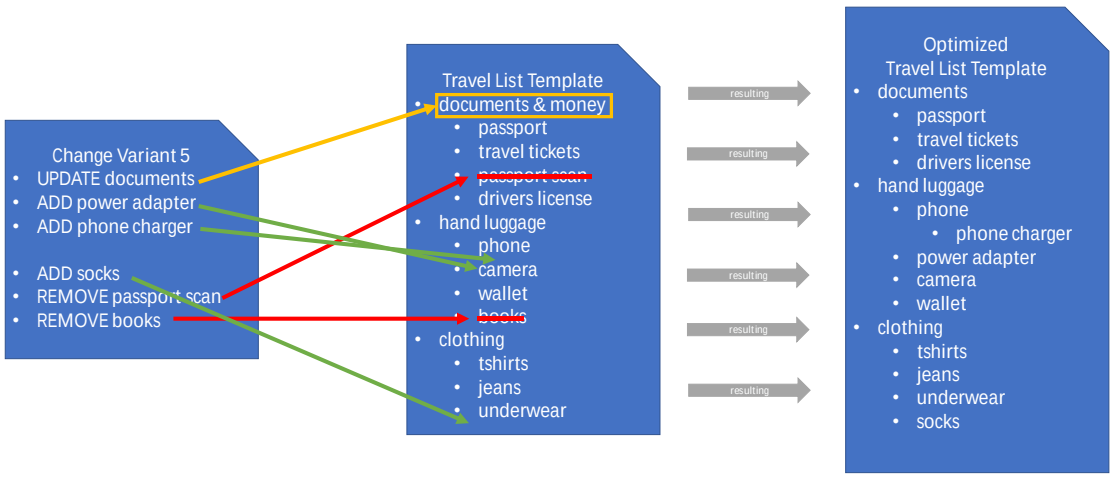


Figure 3.9: Applying Change Variant 5 to the existing Task List Template

3.4 Implementation in a Nutshell

To demonstrate the applicability and technical feasibility of the presented task list lifecycle management approach, it is implemented by various services in the current proCollab protoype (cf. Figure 3.10). A central REST interface (*Lifecycle Service Endpoint*; cf. Figure 3.10) for the services has been implemented so that knowledge workers may use the provided services via the proCollab client.

The *Lifecycle Service* forwards the operation calls from the REST interface to the corresponding sub-services. It represents the centerpiece on the server side and implements the task list template generation pipeline (cf. Chapter 5) and task list template evolution pipeline (cf. Chapter 6) by combining the individual steps from the corresponding services.
The *Similarity Service* includes the implementation of the similarity analysis (cf. Chapter 4). Various methods are provided for determining the individual similarity scores, the similarity matrix, the similarity groups and for the entire similarity analysis.
The *Task List Generation Service* implements all methods required to generate a task list template from similar task list instances. The central component of this service is the cluster mining algorithm.
Methods for the evolution of existing task list templates are provided by the *Task List*

Evolution Service. Furthermore, methods for generating change and execution logs as well as the transformation of change logs into XES format (cf. Section 6.1.1) are provided. Useful methods that are required in various services are implemented centrally in the *Lifecycle Util Service*.
In order to perform the various operations successfully, task list lifecycle management services interact with other proCollab services such as the *Persistence Service*, *Task Service* and *Process Service*. The Task and Process Services provide methods for creating, deleting and updating (task, task tree, and process) instances and templates. The Persistence Service represents the generic interface to the database.

For testing the various functionalities, several JUnit tests (*Task List Lifecycle Tests*; cf. Figure 3.10) were implemented, covering different use cases. For this purpose, multiple use cases are available for the task list template generation and task list template evolution. The tests are triggered via the REST interface and then processed by the server. The testing and validation of the various REST services is accomplished using REST-assured [15].
To test the task list template generation, a new process instance is created at first and several task list instances are created afterwards. There are different test cases, from homogeneous to heterogeneous task lists, for investigating the behavior of the template generation. After the task list instances have been created, the task list template generation pipeline is started.
When testing the task list template evolution, a task list template is required at the beginning. Therefore, in the first step, a task list template for the respective use case is created. The task list template is then instantiated several times and various change operations are executed on the derived task list instances. Once all changes have been made, the task list template evolution is started. As with the task list generation, several use cases are available for the task list evolution.

For the execution of the task list template generation and task list template evolution, the change logs of the respective task list templates or corresponding task list instances have to be analyzed. For this reason, all operations on task lists and tasks need to be logged, both on task list instances and on task list templates. This is realized by the *Access*

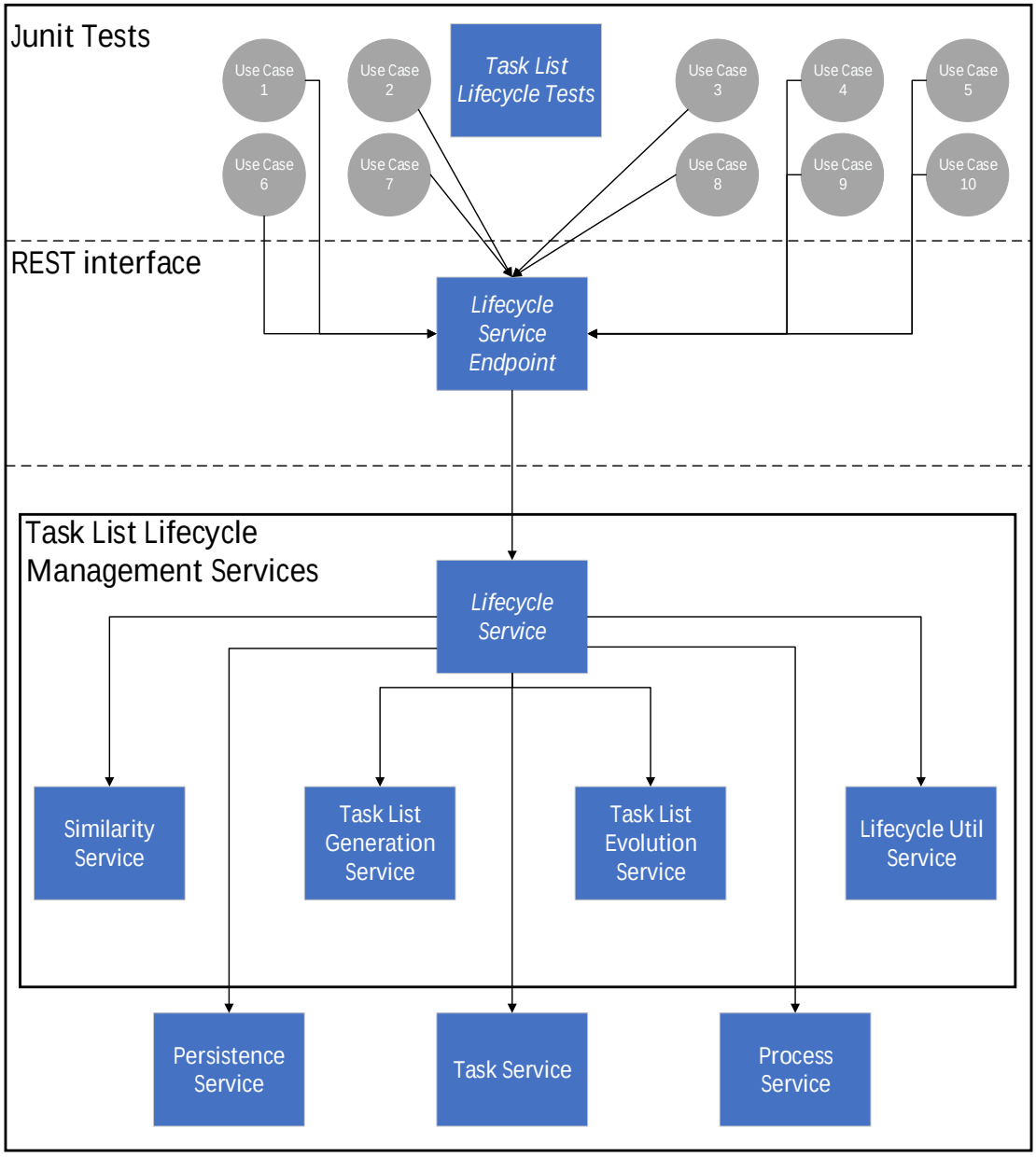


Figure 3.10: Implementation of Task List Lifecycle Management

Control Interceptor. This interrupts a method call and first checks whether a knowledge worker has the necessary authorizations to execute the method. If this is the case, the method execution is continued, otherwise the interceptor throws an error message. After the permissions have been checked, an *Access Control Log Event* with the necessary information (executor, role, timestamp, affected entities, and the parameters of the method call) is created and fired for the corresponding method call. The *Log Service* processes the fired events and creates *Access Control Log Entities* and persists them in the database (cf. Figure 3.11).

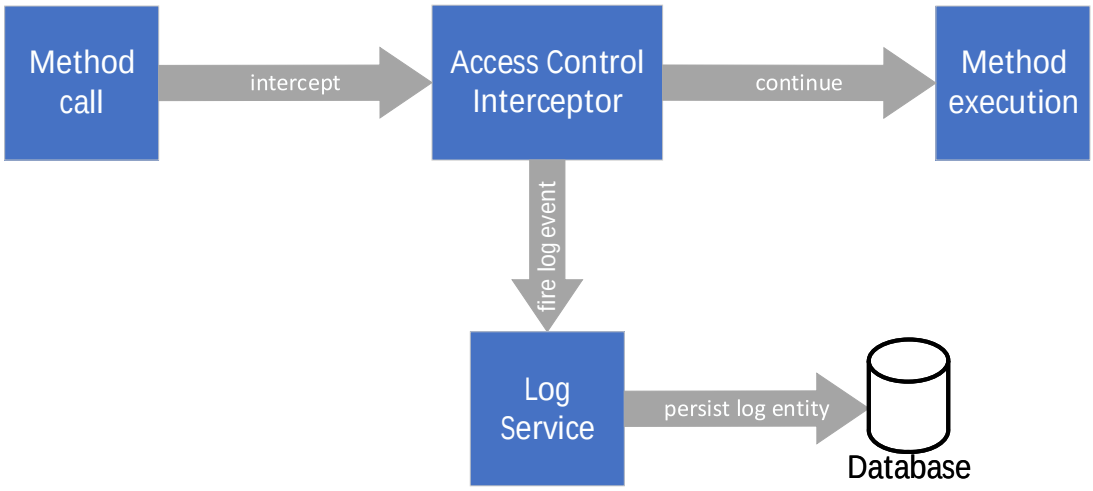


Figure 3.11: Utilization of the AccessControlInterceptor for Change and Execution Logging

4

Similarity Analysis

To enable an appropriate task list lifecycle management, it is important to compare tasks, or change operations on tasks respectively, in order to identify similar tasks. As part of the task list template generation it is necessary to examine the different task list instances for similar tasks to determine the frequencies for similar tasks and to filter out non-frequent tasks. To optimize a task list template using the most frequent change operations on task list instances, change operations have to be compared during the task list template evolution and similar change operations and their frequencies have to be determined.
The difficulty of identifying similar tasks and change operations is discussed in Section 4.1. Subsequently, a first similarity function for the comparison of tasks and change operations on tasks is proposed in Section 4.2. Section 4.3 introduces the similarity matrix, which reflects the similarities between all task pairs. Finally, Section 4.4 explains an approach for the grouping of tasks in similarity groups during the actual similarity analysis.

4.1 Problem Statement

Knowledge workers from different fields often collaborate in a KiP to achieve a common goal by contributing their domain-specific expert knowledge to the KiP. Due to their different fields and backgrounds, the knowledge workers use different vocabularies and terminologies. As a result, the task lists and their respective tasks created or altered by the various knowledge workers differ in their naming and granularity. While a domain expert may handle a complex task as a whole using experience, an inexperienced knowledge worker may need several, simpler tasks to successfully handle a complex task in its entirety. To ensure an automatic and continous generation and evolution of task list templates,

the various tasks need to be analyzed with regard to their similarity. Since there is no adequate similarity function for determining similar tasks, there is a high demand for such a similarity function to enable an appropriate task list lifecycle management.

During the task list template generation, it is necessary to check various task list instances for similar tasks in order to determine their corresponding frequencies and filter out non-frequent tasks. As an assumption for the task list template generation, task list instances are not instantiated using a task list template, but are built from scratch. This increases the difficulty of comparing the individual tasks of the task list instances, since they are not based on a common task list template. To be able to compare tasks and identify similar tasks, an appropriate similarity function, which uses the different attributes of two tasks to determine whether they are similar, is needed.
Due to the uncertainty of KiPs in the form of unforeseen events, changes to the task list instances at run time may be necessary at any time. These changes are made by inserting, removing or updating tasks or task lists. Each of these change operations performed at run time is logged. To identify the most frequent change operations, it is necessary to compare the change operations performed and identify similar change operations. In addition to determining similar tasks, a similarity function has to enable similar change operations to be determined using corresponding log entries.

However, the complexity of natural language makes it extremely difficult to identify similar tasks solely by their naming. Therefore, it is necessary to consider additional information for determining the similarity of tasks. For example, two tasks may have a similar name, but completely different parent tasks. Consequently, a similarity function must consider additional information to ultimately decide whether two tasks or change operations on tasks are similar.

4.2 Similarity Function

Overall, there are different approaches for an appropriate similarity function. A first naive approach is to simply compare the names of tasks and match those tasks with the highest name similarity. In this context, existing activity label matching approaches [16] can be utilized. Furthermore, a common measure to determine the similarity of two strings is the Levenshtein distance [5, 18]. However, the Levenshtein distance only compares text syntactically. Thereby, it misses grammar conjugations and meaningful synonyms. Thus, the matching quality is limited. In turn, the comparison can be significantly improved by splitting texts into a *bag-of-words* [16] first. The similarity score for the bag-of-words representation is then calculated as the number of successfully matched words divided by the total amount of words contained in the larger bag-of-words. As a consequence, the matching of texts containing the same terms, even in a different order, gets more appropriate similarity scores. To further improve the quality of the matching, synonyms may be taken into account when computing the similarity scores. Therefore, a lexical database like WordNet [23] can be deployed.
However, since tasks do not only consist of textual attributes, the similarity function must also consider the remaining attributes to determine the similarity. Figure 4.1 shows different attributes of a task that could be considered for a similarity function.

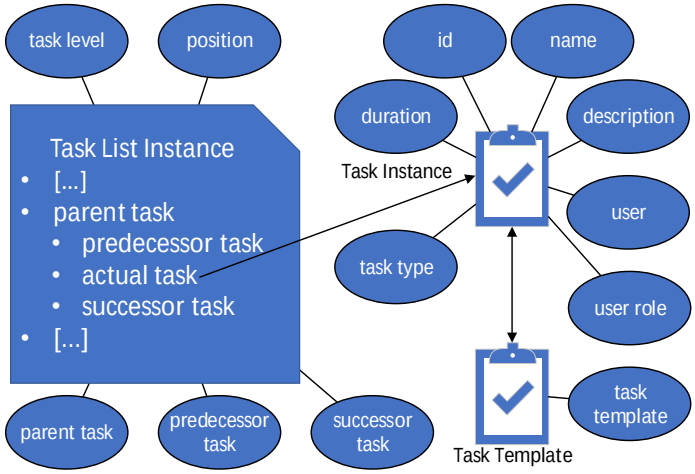


Figure 4.1: Possible Task Attributes for the Similarity Analysis

In addition to the two textual attributes *name* and *description*, the attributes of a task include its *id*, the *duration*, the *task type*, the *user*, the *user's role* and the *task template* from which the task instance was derived. Furthermore, there are the structural attributes *task level*, *position*, *parent task*, *predecessor task* and *successor task*.

Since a task is uniquely identified by its *id*, the *id* is used in the similarity function to check whether two tasks are identical. The two textual attributes *name* and *description* determine what knowledge workers have to do to complete the corresponding task. Therefore, the two attributes are included as the most important attributes in the similarity function. The *duration* of a task is not considered in the similarity function, since the *duration* strongly depends on the knowledge workers involved. For example, an experienced knowledge worker may require significantly less time to complete the same complex task than an inexperienced knowledge worker. As only tasks with same *task type* are comparable, the *task type* must be part of the similarity function. The *user* who created the task is not included in the similarity function. Instead, the *user's role* is taken into account as this allows the comparison of several users with the same responsibilities. Users with the same role may create similar tasks as they are responsible for similar processes and task lists. The *task template* is considered in the similarity analysis, as two tasks derived from the same *task template* are very likely similar. From the structural attributes, only the *task level* is considered in the similarity function, since it indicates whether a task occurs as a parent or a child task. If the *parent element*, the *predecessor* and *successor* of each task were analyzed, this would significantly complicate the similarity determination as the complexity and run time of the similarity analysis would be significantly increased. Therefore, the similarity function only uses attributes that can be efficiently analyzed. In summary, the following attributes are considered in the proposed similarity function for the comparison of tasks and change operations on tasks: *id*, *name*, *description*, *task type*, *user role*, *task template* and *task level*.

To adequately consider the different attributes in a similarity function, a weighted partial similarity is determined for each attribute. The weighting is based on the significance of each attribute for the overall similarity. To determine the partial similarities, it is negligible whether two tasks or two change operations on tasks are compared. If two tasks are

compared, the required attributes are taken directly from the task entities. In contrast, if two change operations are compared, the required attributes are taken from the corresponding change log entries.

4.2.1 Partial Similarities

The partial similarities can be divided into two groups: *simple partial similarities* and *textual partial similarities*.

Simple Partial Similarities

The simple partial similarities include *id*, *task type*, *user's role*, *task template* and *task level*. The respective partial similarity is determined by checking whether the two attributes of the two compared tasks match. The value of the simple partial similarities is either 1 if the attributes match, or otherwise 0. For example, if the ids of the two tasks match, both tasks are equal. If the ids do not match, the other individual partial similarities are used to calculate a similarity score.

Textual Partial Similarities

To calculate the two partial similarities for the attributes *name* and *description*, textual attributes have to be compared with each other in a broader sense than simple equality. Since natural language is very complex, it is very difficult to find a suitable matching for textual attributes. For example, the names of two tasks may contain different terms, which semantically may mean the same, but are presented in different tenses and order resulting in syntactical differences. As already mentioned earlier, there are different approaches for the comparison of strings, such as the Levenshtein distance or bag-of-words representation. These are so-called *string metrics* [4]. In the following, two common string metrics for determining the similarity of two strings are presented: the *q-gram distance* and the *cosine similarity*. The two metrics were selected as they are widely used and based on the bag-of-words concept, thus enabling improved similarity determination.

The *q-gram distance* may be used to determine the similarity of two strings by splitting both strings into so-called *q-grams*. A q-gram consists of exaclty q letters, including spaces, empty characters and special characters. To be able to calculate the similarity of the two strings, the proportion of q-grams present in both strings is determined. The similarity of both strings is calculated using

$$d(a,b) = \frac{2|T(a) \cap T(b)|}{|T(a)| + |T(b)|}. \tag{4.1}$$

The q-gram distance ranges from 0 (i.e. strings have no q-gram in common) to 1 (i.e. strings are equal). For example, the terms $a =$ "*work*" and $b =$ "*wirk*" are divided into the following 3-grams, where *§* represents an empty character:

$$\begin{aligned} T(a) &= \{\S\S w, \S wo, wor, ork, rk\S, k\S\S\}, \\ T(b) &= \{\S\S w, \S wi\S, wir, irk, rk\S, k\S\S\}, \\ T(a) \cap T(b) &= \{\S\S w, rk\S, k\S\S\}. \end{aligned}$$

The final similarity for both terms is $d(a,b) = \frac{2 \cdot 3}{6+6} = 0.5$. Using the q-gram distance, the similarity between strings can be determined, even if they contain spelling errors. Since the name similarity is greater than 0 as long as the two names have at least one q-gram in common, this distance improves the similarity determination even in the presence of spelling errors. For this reason, the q-gram distance is used by the similarity function to determine the name similarity with the assumption that task names consist of only a few words.

In turn, the *cosine similarity* may be utilized for comparing strings by determining term frequencies. For this purpose, the frequencies of the individual terms are determined

and represented as a term frequency vector. The cosine similarity $s(a, b)$ indicates the similarity between two term frequency vectors a and b by calculating

$$s(a,b) = \cos(\phi) = \frac{a \cdot b}{\|a\| \, \|b\|} = \frac{\sum_{i=1}^{n} a_i b_i}{\sqrt{\sum_{i=1}^{n} a_i^2} \sqrt{\sum_{i=1}^{n} b_i^2}}. \tag{4.2}$$

The resulting cosine similarity ranges from 0 (i.e. strings have no common terms) to 1 (i.e. strings are equal) as term frequencies cannot be negative. For example, the names $a =$ "*water bottles*" and $b =$ "*bottles of water*"" are represented by the following term frequency vectors of the form $t = (f_{water}, f_{bottles}, f_{of})$, where f_x represents the frequency of the corresponding term:

$$t_a = (1, 1, 0) \text{ and}$$
$$t_b = (1, 1, 1)\,.$$

The resulting cosine similarity for both strings is $s(a, b) = \frac{1+1+0}{\sqrt{2} \cdot \sqrt{3}} = \sqrt{\frac{2}{3}} \approx 0.82$. The cosine similarity allows to calculate the similarity of two strings without having to consider the order of the words. Therefore, the cosine similarity is well suited to determine the similarity of two strings with a lot of words.
For this reason, the cosine similarity is used by the similarity function to determine the description similarity with the assumption that task descriptions consist of several sentences.

Tasks may be specialized as *to-do items* and *checklist items* based on their task type (cf. Section 2.4.6). To-do items are used for prospective planning, i.e. they are used to define future work. Therefore, the name of a to-do item is formulated as a prompt, e.g. "*purchase 10 bottles of water*" [24]. In turn, checklist items are used for retrospective planning, i.e. they are used to check whether the work to be done was completed. For this reason, the name of a checklist item is expressed as a question, e.g. "*did you buy 10*

water bottles?" [24]. As a result, the names of tasks of the same task type are structured similarly. However, this does not apply to task descriptions, as these are more detailed and usually consist of several sentences. The following explains how the similarly structured names of the tasks are analyzed to determine the name similarity.

Since string metrics only compare the syntactic similarity of two strings, an additional semantic analysis must be performed to adequately compare the task names both syntactically and semantically. For this purpose, the concepts of string metrics and synonym analysis using WordNet are combined. However, a synonym analysis for all terms of two task names is technically not feasible. Therefore, the determination of synonyms is limited to the verbs of the two task names, since the verb is the most descriptive part of a task name. In the following it is assumed that a task name consists of exactly one sentence, which is formulated as a prompt or question.
To identify the different types of words (e.g. verb, noun, etc.), the individual words must be analyzed in the context of the entire sentence. For this purpose, the Stanford Core NLP pipeline may be used [21]. Using the Stanford Core NLP pipeline, each term is annotated with its part-of-speech tag. Synonyms can then be determined for the identified verb by querying the WordNet database for synonym sets. A synonym set contains known synonyms for the corresponding term. If there are intersections between the two synonym sets, the two terms are synonymous with each other. If two synonymous verbs are found, one of the verbs is randomly selected and replaced by the other verb. This ensures that the strings contain the semantically and syntactically identical verb. The synonyms found can thus be taken into account when determining the name similarity. Finally, to determine the name similarity of the two strings, the q-gram distance is used (cf. Equation (4.1)).

Assuming that a description usually consists of multiple sentences and is more detailed than a task name, it is impossible to consider synonyms when determining the description similarity. For this reason the cosine similarity (cf. Equation (4.2)) is used for the determination of the description similarity.

4.2.2 Similarity Score

The final similarity score determining the similarity of two tasks is composed of the invidual partial similarities. Each partial similarity is weighted based on their impact on the final similarity score (cf. Listing 4.1). The following proposed weighting of the individual partial similarities represents a first draft for a weighted similarity function. Most impact on the final similarity score is exerted by the task's name, as the name is the most distinctive attribute briefly describing the work a knowledge worker has to perform to successfully complete the task. As a consequence, the *name similarity* influences 30% of the overall similarity score. The two second most influencing partial similarities are the *description similarity* and the *template similarity*. They are both weighted with 25% of the overall similarity. A task is strongly defined by its description, since it explains in detail what the user needs to do in order to execute the task, but it is not as expressively comparable like the task name. Further, if two tasks are derived from the same task template, it is likely that they are similar to each other. Therefore, the *task template similarity* is strongly weighted, too. The *task type*, *task level* and *role similarities* share the remaining 20% impact on the overall similarity. When the task types do not match at all, the tasks are likely to be different. Consequently, the task type similarity is weighted with 10%. The task level and user role only have a minor influence on the similarity of tasks and are weighted with 5% each.

```
1  double similarity = (30*nameSimilarity
2  + 25*descrSimilarity + 25*templateSimilarity
3  + 10*taskTypeSimilarity + 5*taskLevelSimilarity
4  + 5*roleSimilarity) / 100;
```

Listing 4.1: Similarity Score

The proposed similarity function may be used to determine both the similarity of two tasks and the similarity of two change operations on tasks. In both cases, the similarity is determined using the same attributes. The similarity of two tasks is calculated by directly comparing the two task entities. As change operations performed are consistently logged, the similarity of two change operations may be determined by using the respective change log entries.

4.3 Similarity Matrix

Since a large number of tasks are compared by the similarity function in the course of the similarity analysis, the similarity score calculated for each task pair must be stored in a suitable form. For this purpose, the *similarity matrix* is introduced. A similarity matrix is a 2-dimensional array holding all similarity scores computed by the similarity function. Each row and column corresponds to a respective task. An example similarity matrix is depicted in Table 4.1.

Table 4.1: Example Similarity Matrix

	Task A	**Task B**	**Task C**	**Task D**
Task A	1.0	0.3	0.8	0.5
Task B	0.3	1.0	0.6	0.2
Task C	0.8	0.6	1.0	0.4
Task D	0.5	0.2	0.4	1.0

To fill the similarity matrix, the similarities between all possible task pairs are computed and stored in the matrix. On the main diagonal the similarity score is always 1.0 as *similarity(Task A, Task A)* compares a task with itself and always yields a score of 1.0. As the similarity matrix is symmetrical, *similarity(Task A, Task B) = similarity(Task B, Task A)* applies. Consequently, only the upper triangular matrix has to be analyzed in the similarity analysis to determine similar tasks. During the calculation of the similarity matrix the maximum similarity score is determined. It is used together with a tolerance weight to calculate a threshold value for the similarity analysis. The tolerance weight can either be set individually by the user or a default value (e.g. 30%) is used. If the similarity score of the two tasks exceeds the threshold, the tasks are considered equal. For example, a tolerance weight of 30% is given. The maximum similarity score is 1.0. Consequently, the threshold value is calculated as follows: $threshold = maxSim - toleranceWeight = 1.0 - 0.3 = 0.7$. For two tasks to be considered similar, their similarity score must be at least 0.7.

Listing 4.2 shows the implementation of the similarity matrix calculation. Since the matrix is symmetrical, it is sufficient to calculate only the upper triangular matrix (see loops in lines 6 & 8). For every task pair, the similarity score is calculated (see line 11) and compared with the current maximum similarity (see line 13). Then, the similarity score for

the correspondig task pair is stored in the matrix (see line 15). Once all task pairs have been checked, the threshold for the similarity analysis is calculated (see line 19). Finally, a new similarity matrix is created and returned (see line 20).

```
1  private SimilarityMatrix determineSimilarityMatrix(
2    List<VectorizedTask> tasks, double tolerance) {
3      double maxSim = Double.MIN_VALUE;
4      int size = tasks.size();
5      double[][] simMatrix = new double[size][size];
6      for(int i = 0; i < size; i++) {
7          VectorizedTask taskI = tasks.get(i);
8          for(int j = i; j < size; j++) {
9              VectorizedTask taskJ = tasks.get(j);
10             // determine similarity for task pair
11             double sim = determineSimilarity(taskI, taskJ);
12             // check for max sim
13             maxSim = checkMaxSim(sim);
14             // set similarity score
15             simMatrix[i][j] = simMatrix[j][i] = sim;
16         }
17     }
18     // calculate threshold
19     double threshold = computeThreshold(maxSim, tolerance);
20     return new SimilarityMatrix(simMatrix, tasks, threshold);
21 }
```

Listing 4.2: Implementation of Similarity Matrix Determination

4.4 Similarity Groups

The main goal of the *similarity analysis* is to identify and group similar tasks (cf. Figure 4.2). A similarity matrix holding all similarity scores for all possible task pairs is used as input for the similarity analysis to identify and combine similar tasks into groups. These groups containing similar tasks are therefore called *similarity groups*. One task serves as a representative for the other tasks of the same similarity group.

The identification and grouping of similar tasks is achieved by employing a cluster algorithm. However, there are many different cluster procedures with different approaches. For this

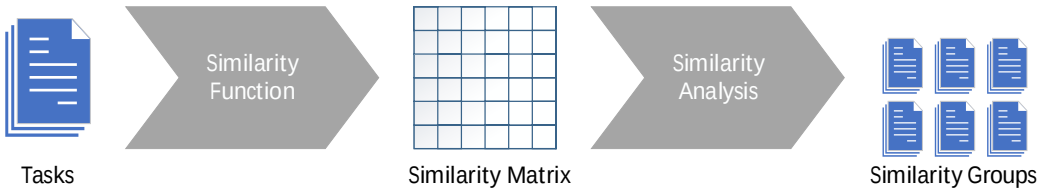


Figure 4.2: Similarity Analysis

reason, two different cluster algorithms were tested for their applicability for grouping similar tasks. In a first attempt, the widely used *k-means* cluster algorithm was used to identify and group similar tasks [20, 39]. However, this approach is problematic as the k-means cluster algorithm requires the parameter k to be set in advance due to the fact that it computes exactly k clusters. As the number of possible similarity groups varies and is unknown, this approach is not suitable for the problem of finding such similarity groups. In a further attempt, the *affinity propagation* [11] clustering algorithm was evaluated. In contrast to the k-means algorithm, affinity propagation does not create a pre-defined number of clusters. A cluster identified by affinity propagation has a representative task– the *exemplar*. Affinity propagation determines its clusters in a number of iterations by passing messages between data points to update two matrices:

- The "responsibility" matrix R has values $r(i, k)$ that reflect how well-suited data point k is to serve as the exemplar for data point i, taking into account other potential exemplars for point i.
- The "availability" matrix A contains values $a(i, k)$ that represent how "appropriate" it would be for data point i to select data point k as its exemplar, taking into account other points' preference for data point k as an exemplar.

Messages are exchanged between data points until an appropriate set of exemplars and corresponding clusters gradually emerges. Two different java implementations were used for clustering similar tasks using affinity propagation [2, 37]. However, it was not possible to reliably cluster similar tasks with either implementation. The quality of the determined clusters varied considerably from one execution of affinity propagation to another. In some cases, all tasks were clustered in a single similarity group in one execution and in

separate similarity groups per task in the next. It is important to note that the tasks were not changed between the executions of the affinity propagation.

Consequently, a different and simpler approach to analyze the similarity matrix and to identify similarity groups is introduced in Listing 4.3. Tasks already belonging to a similarity group are to be skipped and are therefore stored in a set (see line 8). For every task, it is first checked whether it is already in a similarity group and should, therefore, be ignored (see lines 11–13). If it is not contained in a similarity group, a new similarity group is created (see line 15). For every other task, which is not contained in the ignore set, it is checked whether the similarity score between both tasks is greater than the threshold (cf. Section 4.3). If the similarity score exceeds the threshold, the tasks are classified as similar and the other task is added to the current similarity group as well as the ignore set (see lines 23–25). As soon as all other tasks have been analyzed, an exemplar for the respective similarity group is determined and the similarity group is added to the result list (see lines 30 & 31). At the end, the result list is returned (line 34).

Similarity groups are formed on the basis of similarity scores. Therefore, it is possible for two tasks to be combined in the same similarity group, although they are only syntactically similar. Thus, it is important to give the user the possibility to check the similarity groups and, if necessary, make changes. The similarity groups may be reviewed within the task list template generation or task list template evolution, but in any case, after the similarity analysis has been performed. This ensures that the similarity groups meet user expectations by only grouping similar tasks.

```
private List<SimilarityGroup> analyzeSimilarityMatrix(
  SimilarityMatrix similarityMatrix) {
    List<SimilarityGroup> similarityGroups = new ArrayList<>();
    List<VectorizedTask> tasks = similarityMatrix.getTasks();
    int size = tasks.size();

    // tasks that are part of a similarity group
    Set<Integer> ignoreIndices = new HashSet<>();
    for(int i = 0; i < size; i++) {
        // current index part of similarity group?
        if (ignoreIndices.contains(i)) {
            continue;
        } else {
            // create a new similarity group
            SimilarityGroup simGroup = new
               SimilarityGroup(tasks.get(i));
            // other task part of similarity group?
            for (int j = 0; j < size; j++) {
                // skip already assigned tasks and diagonal
                if (i == j || ignoreIndices.contains(j)) {
                    continue;
                } else {
                    // sim(i, j) >= threshold?
                    if (similarityMatrix.getMatrixCellAt(i, j)
                       >= similarityMatrix.getThreshold()) {
                        simGroup.addTask(tasks.get(j));
                        ignoreIndices.add(j);
                    }
                }
            }
            // find exemplar for created similarity group
            simGroup.findExemplar();
            similarityGroups.add(simGroup);
        }
    }
    return similarityGroups;
}
```

Listing 4.3: Implementation of Similarity Group Analysis

5

Task List Template Generation

This chapter presents an approach to automatically derive a common TTT for a group of comparable, completed TTIs. Section 5.1 provides an overview of the task list template generation pipeline. In Section 5.2 the cluster mining algorithm is introduced and its functionality is explained.

5.1 Task List Template Generation Pipeline

The *task list template generation pipeline* performs various tasks to automatically derive a TTT from a given set of TTIs (cf. Figure 5.1). It consists of the similarity analysis (cf. Chapter 4) and the cluster mining algorithm (cf. Section 5.2). The similarity analysis is employed to find similar TIs in the different TTIs and group them into similarity groups. The similarity groups are then used in the cluster mining algorithm to calculate *task frequencies*. The latter are finally used to determine the relevant tasks for the task list template generation and to filter out non-relevant tasks due to their low frequencies.

The first step of the task list template generation pipeline is to perform similarity analysis for TIs of the given TTIs (cf. Chapter 4). This involves combining similar and comparable TIs into similarity groups. In the second step, an order matrix is created for each TTI (cf. Section 5.2.1). Both the similarity groups and the order matrices are utilized in the third step to build the aggregated order matrix (cf. Section 5.2.2). In the last step, the cluster mining algorithm is executed (cf. Section 5.2.3). It clusters two blocks into a larger block in each iteration (step 4a), determines the appropriate order relation for the block (step 4b)

and recalculates the aggregated order matrix (step 4c). These steps are repeated until only one last block remains. This block represents the TTT to be generated.

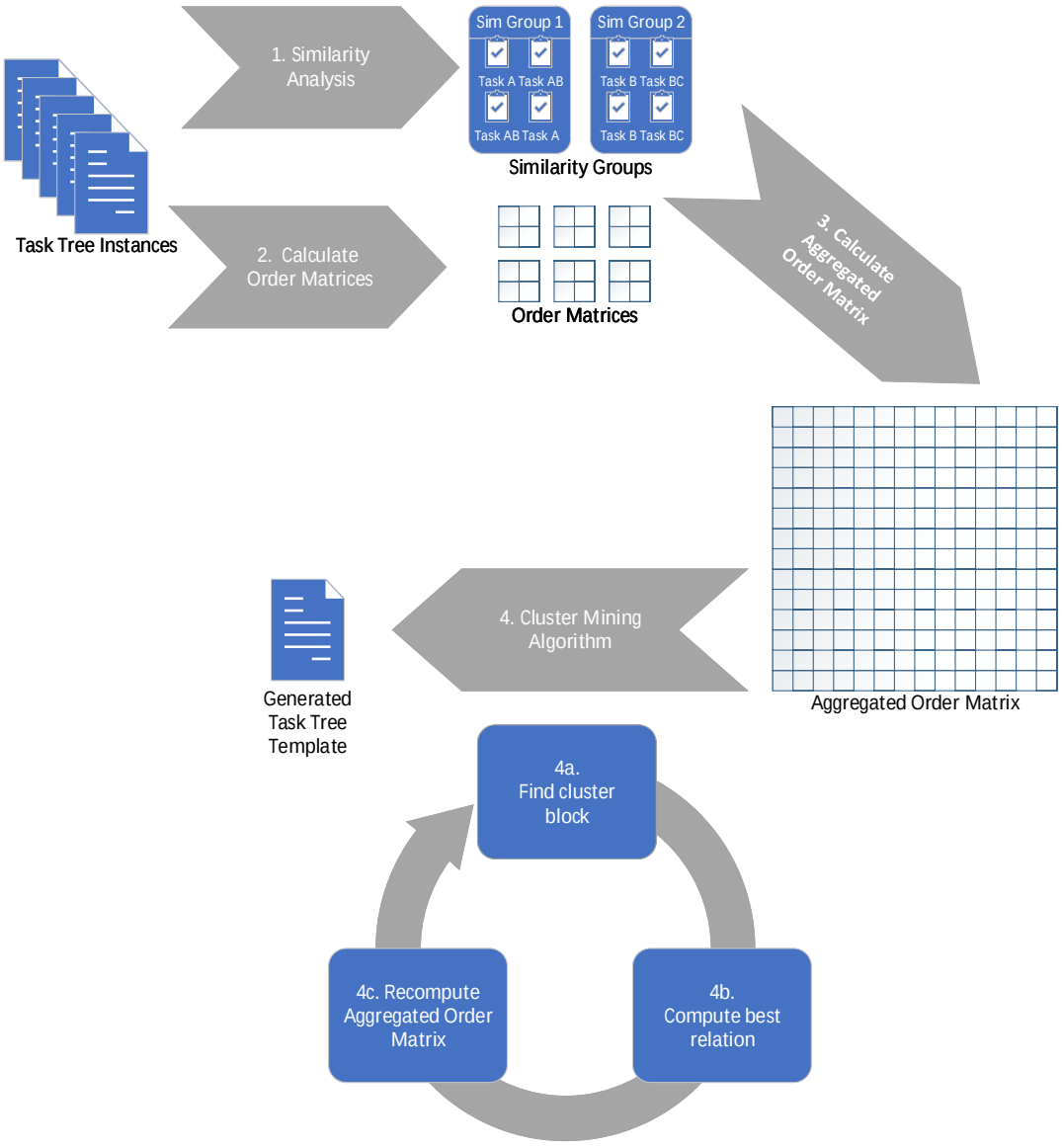


Figure 5.1: Task List Template Generation Pipeline

5.2 Cluster Mining Algorithm

The *cluster mining algorithm* is based on the MinADEPT algorithm [19] for business process variants and was adapted to be applicable to TTIs by [3]. In the course of this thesis, the cluster mining algorithm is adapted to the advanced data model of the second proCollab prototype (cf. Section 2.4.8). Furthermore, the results of the similarity analysis are integrated into the cluster mining algorithm by using the previously identified similarity groups to identify similar tasks when creating the aggregated order matrix.
Listing 5.1 shows the implementation of the cluster mining algorithm. At first, the different relations between TIs are determined for each TTI and recorded in an order matrix (see line 4). Once all TTIs have been analyzed, the identified order matrices are then combined in the aggregated order matrix (see line 7). In each iteration the cluster mining algorithm determines and combines two blocks (see lines 12 - 21). The relations to all other blocks are then determined for the newly created block. As soon as two blocks have been combined, the aggregated order matrix is recalculated (see line 24). When all blocks have been merged into one block, the last block represents the generated TTT (see line 27).

5.2.1 Determining the Order Matrices

To represent the different relations between TIs, each TTI is represented as an *order matrix*. The matrix contains the transitive relations of any TI pair in a TTI. There are two different types of relations for two tasks A and B. Either they are in a predecessor-successor relationship or they are arranged hierarchically in a parent-child relationship, i.e. either A is followed by B (or vice versa) or A is a sub-task of B (or vice versa). If none of the relationships mentioned applies for the tasks, then this is explicitly indicated by the relation type 0 which stands for "no relation". This does not mean that there is no relation at all between these TIs, but definitely none of the above. Table 5.1 lists all possible relation types and their encodings in the order matrix.

To represent a TTI as an order matrix, the relationships of every TI to all other TIs of the given TTI must be analyzed. Furthermore, each child element of a TI is marked as a sub-task (relation type 3). Accordingly, each parent TI is marked as such (relation type 4).

```
1  public TaskTreeTemplate clusterMining(List<TaskTreeInstance>
2    ttis, double minOcc, List<SimilarityGroup> simGroups) {
3      // compute order matrix for every TaskTreeInstance
4      List<OrderMatrix> orderMatrices =
           computeOrderMatrices(ttis);
5
6      // create the aggregated order matrix
7      AggregatedOrderMatrix aggregatedOrderMatrix = new
           AggregatedOrderMatrix(orderMatrices,minOcc, simGroups);
8      int numberOfTasks =
           aggregatedOrderMatrix.getNumberOfBlocks();
9
10     // determine the clusterblocks and then recompute the
           AggregatedOrderMatrix
11     for(int i = 1; i <= (numberOfTasks - 1); i++) {
12       ClusterIndex cluster = new ClusterIndex(0, 1);
13
14       // when the matrix has dimension 2 we can no longer
             compute the separation table and simply have to
             cluster the last 2 remaining blocks
15       if (aggregatedOrderMatrix.getValidDimension() != 1) {
16         cluster =
               getClusterWithMaxBlockLevel(aggregatedOrderMatrix);
17       }
18       // compute cohesion between tasks/blocks to be clustered
19       MatrixCell clusteredBlock =
             aggregatedOrderMatrix.getMatrixCellAt(cluster.idxA(),
             cluster.idxB());
20
21       Cohesion bestRel = computeCohesion(clusteredBlock);
22
23       // recompute aggregated order matrix
24       TaskTreeInstance clusterBlock =
             aggregatedOrderMatrix.recompute(cluster, bestRel);
25     }
26     // last block represents the TTT
27     TaskTreeTemplate result =
           aggregatedOrderMatrix.getResultType();
28     return result;
29 }
```

Listing 5.1: Implementation of Cluster Mining Algorithm

Table 5.1: Possible Relation Types in an Order Matrix

Relation	Meaning
0	no relation between A and B
1	A precedes B
2	A follows B
3	A is sub-task of B
4	A is parent of B

To identify predecessor-successor relations, only TIs with the same parent element are analyzed (relation types 1 & 2). The relations to all remaining TIs can be set to relation type 0 consequently.
An exemplary order matrix for TTI *camping holidays 2* from Section 3.2 is shown in Table 5.2. The order matrix captures the relations between all TIs of this TTI. For instance, *tshirts* is a predecessor of *jackets* (see green cells) and both TIs are children of *clothing* (see yellow cells).

5.2.2 Building the Aggregated Order Matrix

Since an order matrix only contains information about a single TTI, it is necessary to consolidate the information in a superior order matrix. The latter is called *aggregated order matrix* because it represents all TTIs and aggregates their information. In contrast to a cell of a simple order matrix, a matrix cell of the aggregated order matrix contains a 5-dimensional vector denoting the relative frequency of each relation for every pair of TIs in all TTIs as depicted in Table 5.3.

To initially build the aggregated order matrix, three steps must be executed. The different steps are shown in Figure 5.2. First, infrequent tasks, i.e. tasks that occur less frequently than a certain threshold, are filtered out (cf. Section 5.2.2). The average task levels of all TIs are then calculated (cf. Section 5.2.2). Finally, the frequencies of the different relation types are determined for each TI pair and stored in a matrix cell (cf. Section 5.2.2). The various steps are explained in more detail below.

Table 5.2: Examplary Order Matrix of TTI *camping holidays 2* of Section 3.2

Task	Index
documents & money	0
passport	1
map	2
passport scan	3
carry-on baggage	4
camera	5
wallet	6
phone	7
phone charger	8

Task	Index
notebook and pen	9
clothing	10
socks	11
underwear	12
pants	13
jeans	14
skirts	15
tshirts	16
jackets	17

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	-	4	4	4	1	0	0	0	0	0	1	0	0	0	0	0	0	0
1	3	-	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	3	2	-	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	3	2	2	-	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	2	0	0	0	-	4	4	4	4	4	1	0	0	0	0	0	0	0
5	0	0	0	0	3	-	1	1	1	1	0	0	0	0	0	0	0	0
6	0	0	0	0	3	2	-	1	1	1	0	0	0	0	0	0	0	0
7	0	0	0	0	3	2	2	-	1	1	0	0	0	0	0	0	0	0
8	0	0	0	0	3	2	2	2	-	1	0	0	0	0	0	0	0	0
9	0	0	0	0	3	2	2	2	2	-	0	0	0	0	0	0	0	0
10	2	0	0	0	2	0	0	0	0	0	-	4	4	4	4	4	4	4
11	0	0	0	0	0	0	0	0	0	0	3	-	1	1	1	1	1	1
12	0	0	0	0	0	0	0	0	0	0	3	2	-	1	1	1	1	1
13	0	0	0	0	0	0	0	0	0	0	3	2	2	-	1	1	1	1
14	0	0	0	0	0	0	0	0	0	0	3	2	2	2	-	1	1	1
15	0	0	0	0	0	0	0	0	0	0	3	2	2	2	2	-	1	1
16	0	0	0	0	0	0	0	0	0	0	3	2	2	2	2	2	-	1
17	0	0	0	0	0	0	0	0	0	0	3	2	2	2	2	2	2	-

Table 5.3: Aggregated Order Matrix Cell

Relation	0	1	2	3	4
Frequency	f_{rel_0}	f_{rel_1}	f_{rel_2}	f_{rel_3}	f_{rel_4}

Figure 5.2: Steps to build the Aggregated Order Matrix

Filter Non-Frequent Tasks

When the aggregated order matrix is built up, it is necessary to determine the frequencies of the different TIs, i.e. to check in how many TTIs the same or similar TI occurs. Due to the fact that TIs were already grouped together in similarity groups during the similarity analysis, the identification of similar tasks in the different TTIs is facilitated. Since a similarity group's exemplar is the most representative TI, only the respective exemplars are included in the aggregated order matrix as a representative for the other TIs of a similarity group. For this purpose, the frequencies of the exemplars of a similarity group as well as the frequencies of their similar TIs are determined and combined using

$$f_{abs} = f_{exemplar} + \sum_{TI_i} f_{TI_i}. \tag{5.1}$$

The relative frequency of a TI is calculated using

$$f_{rel} = \frac{f}{\#OrderMatrices}. \tag{5.2}$$

Since frequently occurring tasks are solely to be considered when generating a TTT, tasks whose frequency is lower than a certain threshold value must be filtered out. This threshold is set at the beginning of the task list template generation and may be adjusted by knowledge workers according to the specific use case. If non-frequent tasks are also to be considered, the threshold is set lower. If, on the other hand, a TTT is to be generated solely from very frequent tasks, the threshold is set higher accordingly.

In the example presented in Section 3.2, the threshold was 0.5. The filtered out tasks and their respective frequencies can be seen in Figure 5.3.

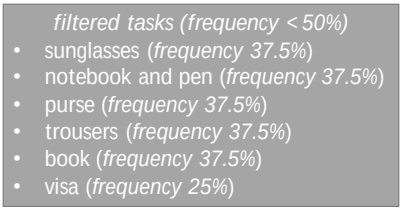


Figure 5.3: Filtered out Tasks for the Use Case of Section 3.2

Determine Average Task Level

The next step of creating the aggregated order matrix is to determine the average task level for each TI that has not been filtered out. The task level starts with task level 0, which corresponds to the elements at the root level of a task tree. Consequently, the task level of child items corresponds to the task level of the parent item increased by one. The task level is important because the cluster mining algorithm first clusters the TIs with the highest task level, i.e. the TIs furthest from the root level. To determine the average task level for a TI, the task level from each order matrix containing the TI is recorded. Then the individual task levels are summed up and divided by the number of order matrices containing the TI.
Table 5.4 shows the average task levels for all tasks that have not been filtered out. Due to the fact that the cluster mining algorithm works with blocks, the task level is called *block level* in the following.

Determine Relations

Finally, the last step in building the aggregated order matrix is to fill the matrix itself. A matrix entry contains a vector with the relative frequencies of the different relation types from all order matrices for a specific TI pair. If the corresponding TI pair has a certain

Table 5.4: Average Task Levels for the Use Case of Section 3.2

Task	avg. Task Level
documents & money	0.0
passport	1.0
travel tickets	1.0
drivers license	1.0
map	1.0
carry-on-baggage	0.0
camera	1.0
wallet	1.0
phone	1.0
phone charger	1.0
power plug adapters	1.0
Europe	2.0
USA	2.0
Asia	2.0
clothing	0.0
socks	1.0
underwear	1.0
undergarments	1.0
jeans	1.0
pants	1.0
skirts	1.0
shirts	1.0
tshirt	1.0
jackets	1.0
leather jacket	1.75
sports coat	1.75

relation type in an order matrix, the value of the 5-dimensional vector for this relation type is increased by $\frac{1}{n}$, where n represents the number of order matrices.
Table 5.5 lists the order relations between *jackets* and *leather jacket* in all TTIs from the example in Section 3.2 and Table 5.6 shows the corresponding entry in the aggregated order matrix.

Table 5.5: Order Relations between *jackets* and *leather jacket* in all TTIs of Section 3.2

TTI	Frequency	Order Relation
beach holidays 1	0.125	4
beach holidays 2	0.125	1
beach holidays 3	0.125	4
hotel holidays 1	0.125	4
camping holidays 1	0.125	0
camping holidays 2	0.125	0
hiking holidays 1	0.125	0
hiking holidays 2	0.125	0

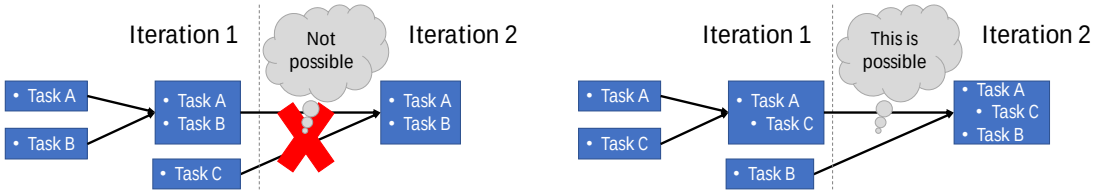
Table 5.6: Resulting Aggregated Order Matrix Cell for *jackets* and *leather jacket*

Relation Type	0	1	2	3	4
Frequency	0.5	0.125	0.0	0.0	0.375

5.2.3 Determining the Cluster Block

Once the aggregated order matrix has been built up, the cluster mining algorithm is started. As it forms a new *cluster block* from two cluster blocks (or TIs initially) in every iteration, the cluster mining algorithm requires n-1 iterations to generate a TTT candidate. The algorithm starts with the blocks exposing the highest block level, i.e. it works bottom-up. The reason for this procedure is the structure of task trees: if TIs with low block levels were clustered first, it would be impossible to map hierarchical relations correctly. For example, two TIs "Task A" and "Task B" are clustered together as a cluster block with the relation *"Task A precedes Task B"*. In the next step "Task C" should be added as a sub-task of this cluster block. Because it is unknown whether "Task C" should be a sub-task of "Task A" or "Task B", the correct hierarchical relation cannot be displayed. But if, at first, a cluster block with the relation *"Task C is a sub-task of Task A"* is created, it is no problem to map

the relation *"Task A precedes Task B"* in the next step.
Figure 5.4 illustrates both different cases: (a) shows why the clustering is not possible when TIs with lower block level are clustered first and (b) shows why TIs with a high block level are clustered first.



(a) Clustering not possible (b) Clustering possible

Figure 5.4: Bottom up Clustering Approach

In this context, the average block level is the most important selection criteria to find a suitable cluster block. When two blocks (or TIs initially) are clustered together and build a new cluster block, its average block level is determined using

$$avgBlockLevel_{neu} = \frac{avgBlockLevel_{old_1} + avgBlockLevel_{old_2}}{2}. \tag{5.3}$$

First, the average block level of all possible block pairs is calculated. Then, the block pairs are sorted in descending order by their average block levels and the block pair with the highest block level is selected. If there are several candidates with the same block level, the so-called *separation value* is used as the second selection criteria. The separation value expresses the similarity of the order relations of two blocks in comparison to the other blocks. The higher this value is, the better the two blocks are suited for being clustered. Hence, the block pair with the highest separation value is selected and finally clustered. However, if there were several block pairs with the same separation value, a block pair would be selected randomly. In this case, the third selection criteria to select a block pair is the so-called *cohesion value* [19]. It expresses the significance of an order relation for the pair of blocks to be clustered. To calculate the cohesion value for a block, it is necessary to first determine the order relation that is most similar to the block's vector v_b, i.e. the

respective entry for the block in the aggregated order matrix. The closeness, or similarity, of two vectors $\alpha = (x_1, x_2, \ldots, x_n)$ and $\beta = (y_1, y_2, \ldots, y_n)$ can be calculated using

$$f(\alpha, \beta) = \frac{\alpha \cdot \beta}{|\alpha| \times |\beta|} = \frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} x_i^2} \times \sqrt{\sum_{i=1}^{n} y_i^2}}. \tag{5.4}$$

To be able to calculate the closeness between the block vector and the five order relations, each order relation is represented by a unit vector: $v^0 = (1, 0, 0, 0, 0)$, $v^1 = (0, 1, 0, 0, 0)$, $v^2 = (0, 0, 1, 0, 0)$, $v^3 = (0, 0, 0, 1, 0)$ and $v_4 = (0, 0, 0, 0, 1)$. The cohesion value for the block pair is calculated using

$$cohesion(v_b) = \frac{\max_{\diamondsuit=\{0,1,2,3,4\}} f(v_b, v^{\diamondsuit}) - 0.4472}{1 - 0.4472}. \tag{5.5}$$

The value range of $\max_{\diamondsuit=\{0,1,2,3,4\}} f(v_b, v^{\diamondsuit})$ is $[0.4472, 1]$ [19]. Therefore, the cohesion value is normalized into the value range $[0, 1]$. The cohesion equals 1 if there is a dominant order relation for the cluster block (e.g. $(0, 0, 1, 0, 0)$) and 0 if all order relations are equally weighted (e.g. $(0.2, 0.2, 0.2, 0.2, 0.2)$). Finally, the block pair with the highest cohesion value is selected from the cluster candidates with the same separation value.

If a cluster block is found, the relationship in which the two blocks are combined, i.e. whether a hierarchical or a predecessor-successor relationship exists between the clustered blocks, must still be determined. The relation is obtained from the corresponding entry in the aggregated order matrix by selecting the relation type with the highest frequency. If the relation type 0 has the highest frequency, the next best relation will be selected alternatively since two blocks without any given relation cannot be clustered.
Figure 5.5 illustrates the clustering step for the two blocks *jackets* and *leather jacket & sports coat*. Both blocks have been selected to be clustered. To determine the order relation, the corresponding entry in the aggregated order matrix is utilized. The most frequent order relation, which is not 0, is then used to cluster the two blocks. In the example, the two blocks are clustered with order relation 4, i.e. *jackets* is the parent of *leather jacket & sports coat*.

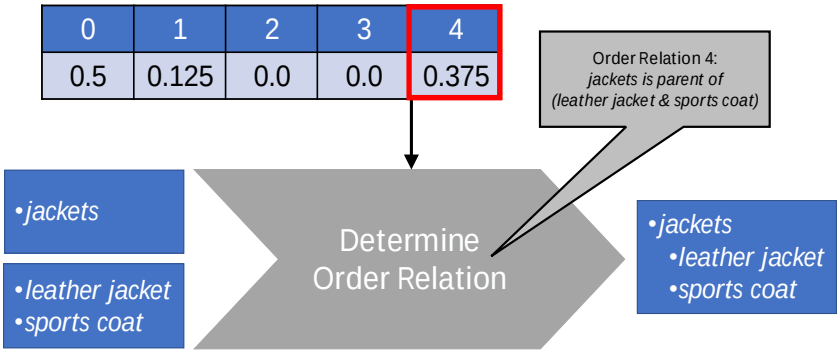


Figure 5.5: Determining the Order Relation for the Cluster Block

5.2.4 Recomputing the Aggregated Order Matrix

Since two blocks are combined into one block during the clustering procedure, the number of blocks in the aggregated order matrix is reduced by one in every iteration. As a result, the aggregated order matrix must be recalculated to determine the relations of the clustered block to all other blocks. This is accomplished by calculating the mean values of the order relations between the clustered blocks and all other blocks.

5.2.5 Cluster Mining Result

After recalculating the aggregated order matrix, the previously described steps, i.e. identifying two blocks to be clustered, determining the order relation within the clustered block, and recalculating the aggregated order matrix, are repeated. In each iteration, two blocks are merged into one larger block. This iterative clustering is repeated until all blocks have been combined in one block. This block then represents the TTT candidate generated from the TTIs.

Figure 5.6 illustrates the sequence of cluster steps for the example from Section 3.2 as well as the resulting TTT. In the first three iterations of the cluster mining algorithm, the blocks with the highest block level, i.e. block level 2, are clustered first. In iterations four and five, the first hierarchical relations are clustered, i.e. the blocks with the highest block level are clustered with their parent blocks. Then, up to the 20th iteration, the blocks with block level 1 are clustered with predecessor-successor relations. In iterations 21, 22 and

23, the subordinated blocks with block level 1 are clustered with the superior blocks with block level 0 in a parent-child relation. In the last two iterations, the root level blocks are clustered with predecessor-successor relations. The resulting TTT candidate is shown on the right.

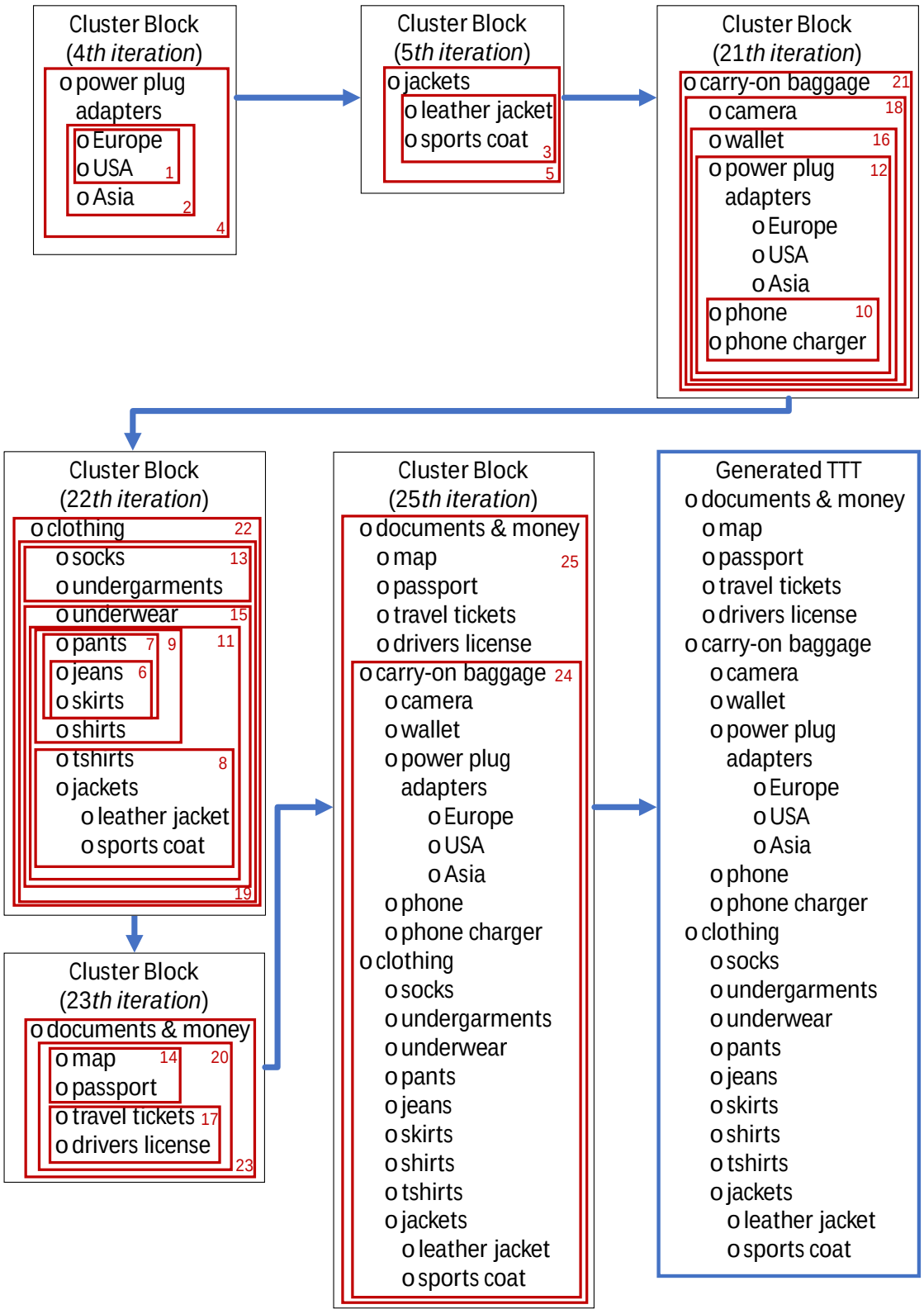


Figure 5.6: Sequence of Cluster Steps for the TTIs of Section 3.2

5.3 Summary

This chapter introduces the task list template generation pipeline in proCollab for generating a common TTT from related TTIs. At first, similar TIs must be identified in the different TTIs and grouped into similarity groups. Each TTI is then converted to an order matrix to represent the mutual relations between the TIs in the corresponding TTI. Subsequently, the order matrices are consolidated in the aggregated order matrix, which combines all relationship information of the TIs from all TTIs. In an iterative clustering approach, the cluster mining algorithm merges two cluster blocks into one larger cluster block in each iteration. After all blocks have been combined in a cluster block, the algorithm terminates. The final cluster block forms the collective TTT candidate for the given TTIs.

In the course of this thesis, the cluster mining algorithm from the first proCollab prototype [3] was adapted to the new, advanced data model of the second proCollab prototype. For this purpose, the merging of two cluster blocks into a new cluster block was adapted. A cluster block is represented by a task tree and has to be built up accordingly. Since the new implementation of task trees utilizes adjacency lists, the elements of a cluster block have to be added with the proper relation to the new cluster block. In contrast to the first prototype, the new data model allows hierarchical relations to be directly mapped using tasks, i.e. a parent task may directly refer to a list of sub-tasks and no additional task tree containing the child elements needs to be inserted.
A further adaptation of the cluster mining algorithm is the improvement of the selection of two blocks to be clustered. In the first implementation of the cluster mining algorithm, the two blocks to be clustered are first selected according to the average block level. If there are several candidates with the same block level, the block pair with the highest separation value will be selected. However, if there are several block pairs with the same separation value, a block pair will be selected randomly. Therefore, the selection of the block pair to be clustered has been extended by taking the cohesion value into account in case there are several candidates with the same separation value. As a result, blocks with more significant order relations are clustered first.
However, the inclusion of a similarity analysis to determine similar tasks is the most important change. This involves analyzing tasks and additional meta information to identify

similar tasks from the various TTIs and grouping them together. The results of the similarity analysis in the shape of similarity groups are used to generate the aggregated order matrix, i.e. to determine task frequencies and to filter non-frequent tasks. By employing a dedicated similarity analysis based on tasks and meta information, the identification of similar tasks is improved compared to the trivial function used in the first proCollab prototype.

However, further enhancements in future work are conceivable. For example, the threshold for filtering non-frequent tasks could be automatically adjusted based on the properties of the given TTIs. Thus, the threshold value for very similar TTIs could be automatically adjusted in such a way that only very frequently occurring tasks are considered in the task list template generation. Accordingly, the threshold could also be adjusted in a similar manner for very heterogeneous TTIs.

6

Task List Template Evolution

This chapter introduces an approach for the automatic optimization and evolution of existing TTTs based on change operations on TTIs. Section 6.1 provides an overview of the task list template evolution pipeline and its functionality. Section 6.2 presents an approach for extending the task list template evolution by analyzing nested insert operations and generating a TTT containing the inserted tasks. Using the generated TTT, the inserted and nested tasks may be used to optimize the original TTT.

6.1 Task List Template Evolution Pipeline

The *task list template evolution pipeline* consists of five major steps (cf. Figure 6.1). First the change logs are pre-processed (cf. Section 6.1.1), then a process mining algorithm is applied to the logs to mine the *change process* (cf. Section 6.1.3) and to determine different *change process variants* (cf. Section 6.1.4). Subsequently, the most frequent change operations are determined and *change variants* are generated (cf. Section 6.1.6). Finally, one or several change variants are selected by knowledge workers in order to *update* the original TTT or to *replace* the original TTT with multiple specialized TTTs or to *add* additional specialized TTTs (cf. Section 6.1.6).

6.1.1 Pre-Processing of the Change Logs

To optimize a TTT, it is important to select the corresponding change logs, since every change operation applied to TTIs at run time is recorded in such logs. First, all change logs

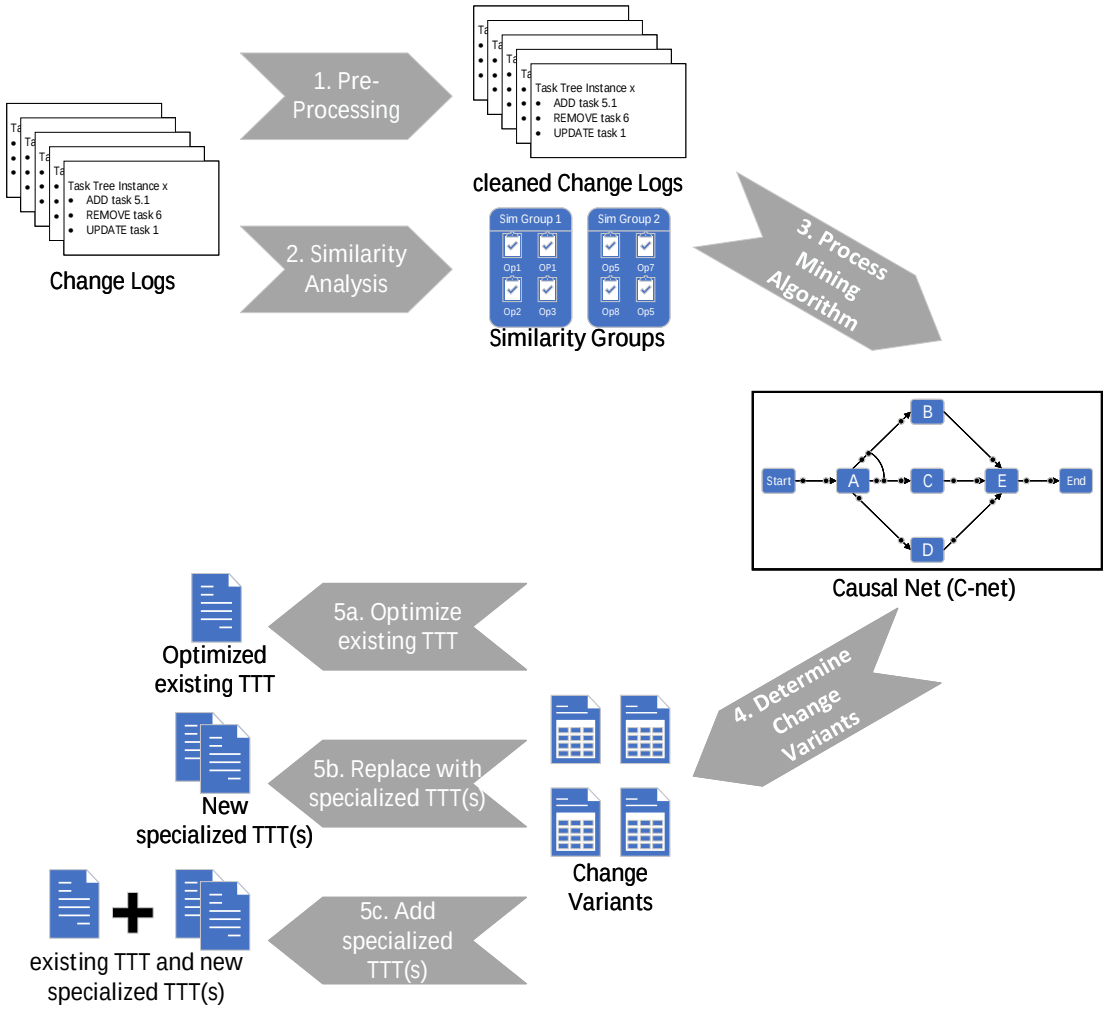


Figure 6.1: Task List Template Evolution Pipeline

of TTIs instantiated from the TTT are selected. If all change logs were used for optimization, this would result in a general optimization of the TTT. However, the set of change logs and, thus, the set of considered TTIs can be limited as input for the optimization. The number of change logs can be limited by selecting only the change logs of the latest instantiated TTIs. This allows the most recent changes to be included in the existing TTT each time the task list template evolution is executed. As a result, the TTT evolves with each execution of the task list template evolution (cf. Figure 6.2).

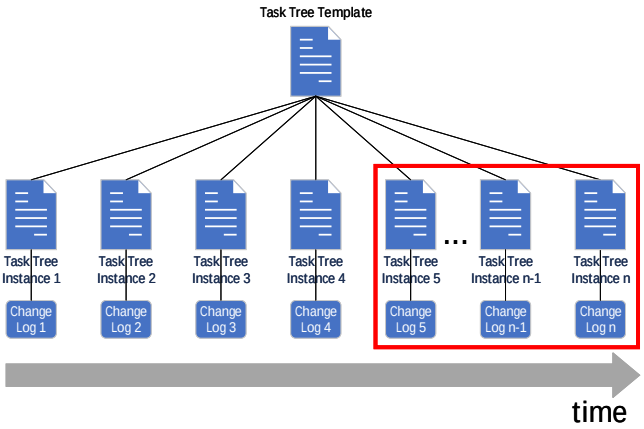


Figure 6.2: Only the latest Change Logs are selected for Optimization

Furthermore, a decision has to be made regarding the number of change log entries to be considered. Although it is possible to perform the optimization with all contained log entries, it may be more useful to use only a subset of the entries. For example, if knowledge workers need to customize the derived TTI in a similar way each time they instantiate a TTT, it may be useful to select only change operations that were applied to the corresponding TTI immediately after the instantiation (cf. Figure 6.3). By integrating these changes into the existing TTT, knowledge workers could be better supported, as they would not have to adapt the TTT themselves.

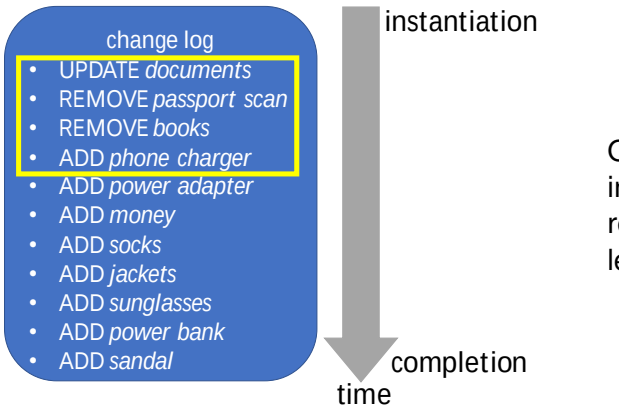


Figure 6.3: Only a Subset of Change Operations is selected for Optimization

Obviously, change logs may contain changes that were accidentally made and then undone. To prevent such changes from being included in the optimization, the change logs should first be cleaned by removing these entries.
In this context, note that only the basic operations (*insert*, *delete* and *update*) are considered in the following. This does not limit the applicability of the presented approach as discussed in Section 2.5. Any high-level change operation on task trees may be expressed as a sequence of basic operations (cf. Figure 2.15).

The final step in pre-processing the change logs is to convert the change logs to the XES (extensible event stream) format [1]. This step is necessary because the used process mining algorithm receives change logs in XES format as input. The XES IEEE standard aims to provide a uniform and extensible format for recording system behavior using logs. An XES change log consists of various *traces* representing the individual change logs of the TTIs. A trace contains the change operations of the corresponding change log as events. The hierarchy of the different XML tags is depicted in Figure 6.4.

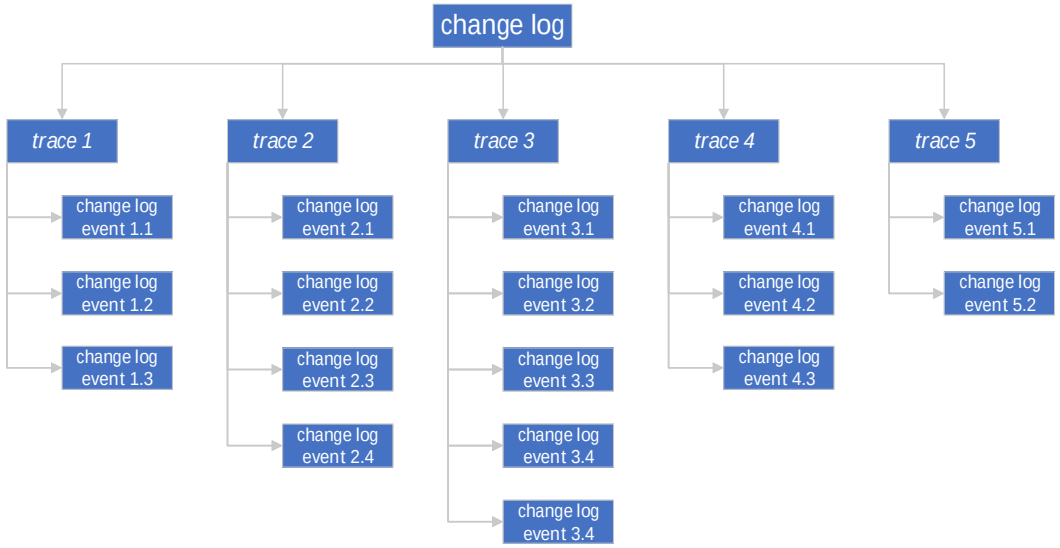


Figure 6.4: XES Structure

Each trace is represented by the id of the underlying TTI. For each *event*, the system logs *who* made the change, *which* element was changed, *how* it was changed (insert, delete, update) and *when* it was changed. The XLog (change log in XES format) serves

as a starting point for further analysis. Figure 6.5 summarizes the individual steps of pre-processing the change logs once again: first the logs are cleaned and then converted to the XES format.

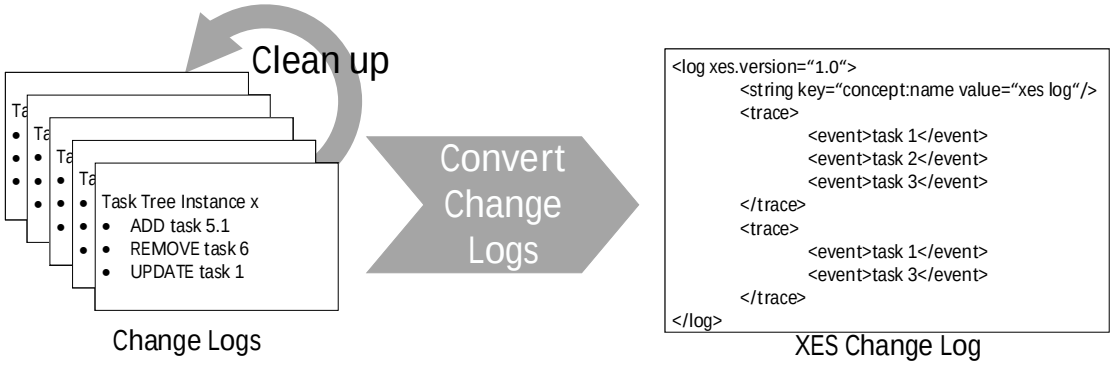


Figure 6.5: Pre-Processing of Change Logs

6.1.2 Determining Similarity Groups

The next step in the task list template evolution pipeline is the similarity analysis (cf. Chapter 4) of the change logs. Since multiple change operations may be applied to TTIs at run time by knowledge workers, it is important to analyze the similarity of these change operations. Similar change operations from the various change logs are combined into similarity groups during the similarity analysis to enable the most frequent change operations to be identified in a later step (cf. Section 6.1.5). To compare the change operations, they are grouped by operation type (insert, remove, update) and then similarity groups are determined for each operation. This creates three similarity groups: one group for insert operations, one for remove operations and one for update operations. This way, only change operations of the same type are compared. This approach is sufficient because two change operations with different types can never be the same. Figure 6.6 illustrates the similarity analysis for change logs.

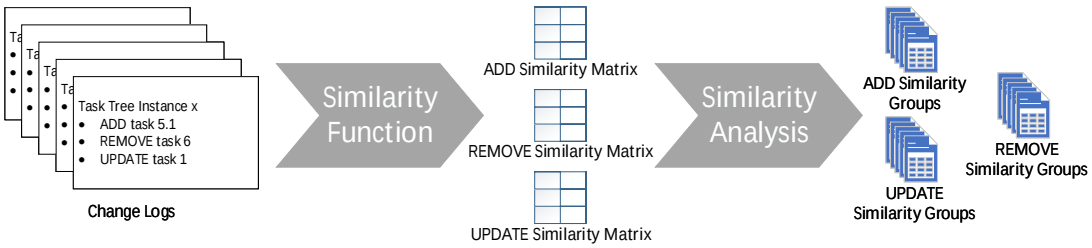


Figure 6.6: Creation of Similarity Groups based on Change Logs

6.1.3 Multi-Phase Miner

A process mining algorithm is used to identify the change processes from the change logs containing the history of change operations for the selected TTIs. For this purpose, the *multi-phase mining algorithm* proposed by [6, 7] is used as it combines the change operations from the different change logs and their causal relations among each other in a C-net (cf. Section 2.6.2) over multiple phases. The resulting C-net represents the *change process*, which was initiated by knowledge workers at run time by applying various change operations to the corresponding TTIs. The C-net may be traversed in a later step to identify possible paths through the C-net (cf. Section 6.1.4). The algorithm receives the change logs in XES format as input and generates a C-net in two steps: first a C-net is created for each trace and then all generated C-nets are combined in one C-net. Figure 6.7 illustrates this process. Each node in the C-net represents a change operation from the corresponding change log. In addition to creating the C-net, the miner builds a map structure that contains the previous changes for each change operation. This map is used later to remove invalid variants as the mining algorithm generalizes the behavior observed in the change logs [46]. As a result of the generalization, the mining algorithm can also allow traces that were not contained in the change logs.

As an illustrative example, the resulting C-net from the use case presented in Section 3.3 is shown in Figure 6.8. The three different change logs are each valid paths through the mined C-net. As can be observed in the C-net, a path is only valid when the change operations *ADD power adapter* and *REMOVE passport scan* or *REMOVE books* have been successfully executed. The reason for this is that the end node only has these two

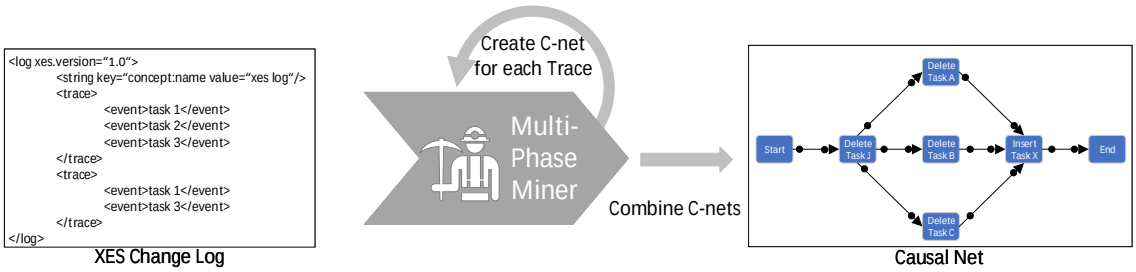


Figure 6.7: Creating the C-net using the Multi Phase Miner

input bindings and is therefore bound to all three nodes, i.e. to be able to execute the end node, one of the two input bindings has to be fulfilled.

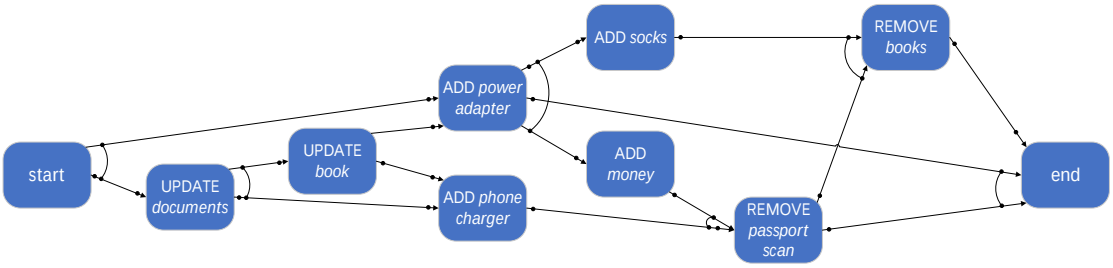


Figure 6.8: Resulting C-net for the Use Case of Section 3.3

The final C-net is an example for *underfitting* since it allows more valid traces than were contained in the original change logs. However, the map with the previous changes created during the mining process allows the identification of the valid change process variants in a further step (cf. Section 6.1.4).

6.1.4 Determining Change Process Variants

A *change process variant* represents a possible path through the C-net and consists of an id and two node sets: *active nodes* (i.e. nodes that have not yet been visited) and *visited nodes*. The set of change process variants can be determined by traversing the C-net in forward direction, i.e. from the start node to the end node. Each output binding of a C-net node represents an alternative path (i.e. an exclusive decision) and must therefore be regarded as a branch to other change process variants. Figure 6.9 illustrates this process.

In this process, it is decisive in which order the output bindings are examined since the output bindings of the C-net are ordered by their frequencies. If the output bindings with low frequencies are processed first, more change process variants will be obtained. However, if the output bindings with high frequencies are examined first, it will result in fewer change process variants. For the following example, the output bindings were sorted in ascending order of frequency, i.e. the low frequency bindings were processed first and as a result more change process variants are determined.

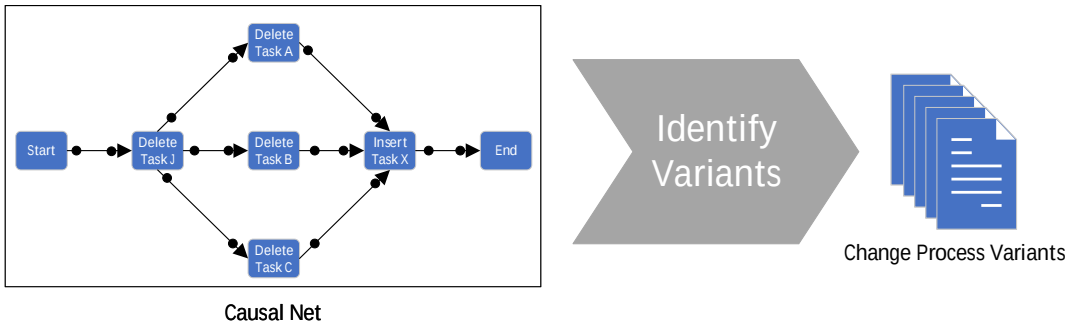


Figure 6.9: Identification of Change Process Variants

Listing 6.1 shows the implementation of the determination of the change process variants. First, a new change process variant is created for each output binding of the start node (see line 4). For each change process variant generated, it is subsequently checked whether there are still active nodes, that is, nodes that have not been visited yet. In case that there are no more active nodes, the change process variant was processed successfully and the end node was reached sucessfully (see line 11). Otherwise, an active node is determined for the current change process variant whose input bindings are fulfilled enabling the node to be visited (see line 13). If there are no more active nodes with fulfilled input bindings, the change process variant will be invalid and be removed (see line 16). The selected active node is marked as visited in the next step (see line 18). Afterwards, the output bindings of the active node are evaluated. The first output binding is added to the current change process variant and a new change process variant is created for each additional output binding (see lines 22 - 26). These steps are repeated until all

change process variants have been completely processed or classified as invalid (see lines 6 - 29). At the end, the valid change process variants are returned (see line 30).

```
1  private List<CPVariant> traverseCNet(CNet cNet) {
2      for (CNetBinding startBinding : cNet.getStartBinding()) {
3          // create change process variant for every start
               outputbinding
4          createNewChangeProcessVariant();
5      }
6      while (!variants.isEmpty()) {
7          for (int i = 0; i < variants.size(); i++) {
8              CPVariant variant = variants.get(i);
9              if (variant.getActiveNodes().isEmpty()) {
10                 // variant is valid
11                 addValidChangeProcessVariant(variant);
12             }
13             CNetNode activeNode = variant.getActiveNode();
14             if (activeNode == null) {
15                 // variant is invalid
16                 removeInvalidChangeProcessVariant(variant);
17             }
18             variant.setActiveNodeVisited();
19             List<CNetBinding> bindings =
                   activeNode.getOutputBindings();
20             if(!bindings.isEmpty()) {
21                 // add outputbinding to current change process
                       variant
22                 variant.addOutputBinding(bindings.get(0));
23                 for(int j = 1; j < bindings.size(); j++) {
24                     // create a new change process variant for
                           each other output binding
25                     createNewChangeProcessVariant();
26                 }
27             }
28         }
29     }
30     return validVariants;
31 }
```

Listing 6.1: Implementation of Change Process Variant Detection

Figure 6.10 illustrates the change process variants determined for the use case presented in Section 3.3. Although there were only three change logs, five change process variants were identified. This is because a C-net also enables traces that did not occur in the change log, but are causally possible. During the analysis, a total of 22 change process variants were identified, of which only the five resulting change process variants were valid.

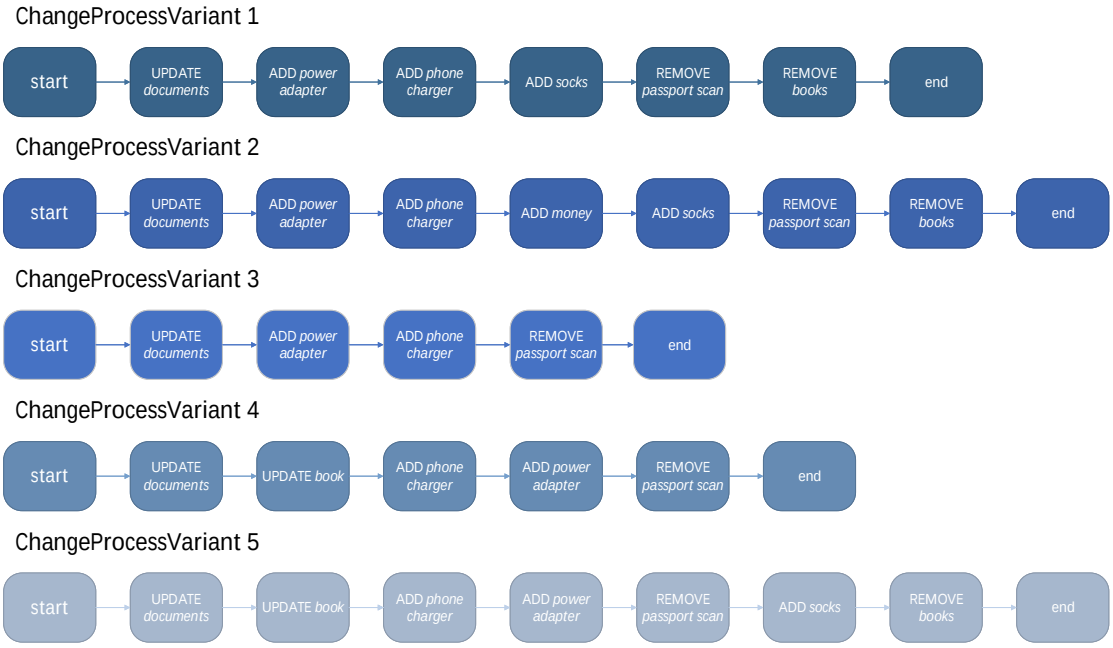


Figure 6.10: Change Process Variants determined for the Use Case of Section 3.3

6.1.5 Identifying Change Blocks

The task list template evolution pipeline continues with the search for *change blocks* after all valid change process variants have been determined. A change block is a sequence of frequently occurring changes. First, a map with the most frequent changes is generated. Changes whose frequency is less than a certain threshold (e.g. 50%) are ignored. To create the map containing the most frequent changes, the frequencies of the change operations are determined in all change process variants and summed up. Table 6.1 contains the absolute and relative frequencies of the individual operations applied to tasks for the Use Case presented in Section 3.3.

Table 6.1: Task Frequencies for the Use Case of Section 3.3

Task	abs. Frequency	rel. Frequency
UPDATE documents	5	1.0
UPDATE book	2	0.4
ADD power adapter	5	1.0
ADD phone charger	5	1.0
ADD socks	3	0.6
ADD money	1	0.2
REMOVE passport scan	5	1.0
REMOVE books	4	0.8

Subsequently, frequently occurring change blocks are searched for by traversing all change process variants. For each change operation, it is verified whether it is a frequent operation or not. If this is a frequent change operation, a change block of maximum length, starting at the position of the currently examined operation, will be created by the algorithm by adding subsequent change operations to the change block when their frequencies are high enough, i.e. when they are frequent operations. When the end of a change block has been reached, the change block is saved in a map or its frequency is updated, in case the change block has already occurred in other variants before.

Figure 6.11 shows the different change blocks contained in the identified change process variants (cf. Figure 6.10). The minimum frequency of a change to be considered in a change block was 50% for the example. Since the *UPDATE book* and *ADD money* change operations occurred less frequently, they were not included in any change block.

Figure 6.11: Change Blocks found for the Use Case of Section 3.3

6.1.6 Determining Change Variants

In the next step, the previously determined change blocks are applied to the original TTT and a *change variant* is created as a result. A change variant represents the original TTT to which one of the change blocks was additionally applied (cf. Figure 6.12).

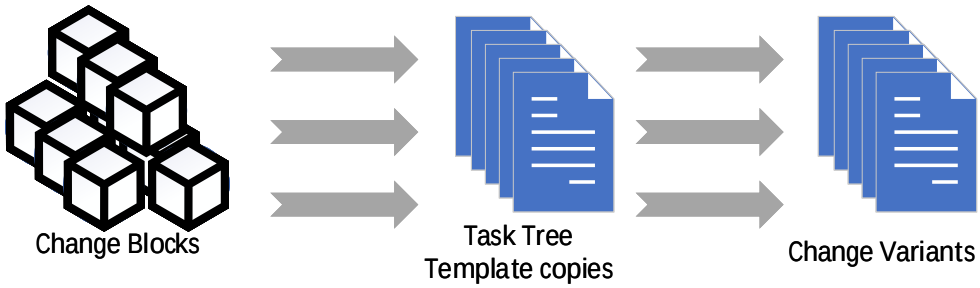


Figure 6.12: Apply Change Blocks to copies of the original Task Tree Template

Listing 6.2 shows the implementation of the determination of change variants. In the first step, a change variant is created for every change process variant (see line 7). Subsequently, a copy of the original TTT is created for each change block (see line 13). Then the change block, i.e. all change operations contained in the corresponding change block, is applied to the TTT copy (see line 15). In the last step, the applied change operations and the optimized TTT are added to the change variant (see line 17). Finally, each change variant is added to the result list (see line 19).

Figure 6.13 shows the different identified change variants for every identified change process variant (cf. Figure 6.10). Change variants 2, 4 and 5 each contain two change blocks, whereas change variants 1 and 3 each contain only one change block.

Afterwards, it is checked whether the generated change variants contain exactly the same change ooperations (i.e. same tasks, same position, same parent element, etc.). If there are two change variants matching each other, one of them will be deleted. Subsequently, the knowledge workers have the possibility to select one or more change variants to optimize the existing TTT or to add new, specialized TTs (cf. Figure 6.14).

```
1  private List<ChangeVariant>
       applyChangeBlocks(TaskTreeTemplate original,
       List<ChangeBlock> changeBlocks) {
2      Map<Integer, List<ChangeBlock>> variantIdToChangeBlocks =
           mapVariantIdToChangeBlocks(changeBlocks);
3      List<ChangeVariant> resultList = new ArrayList<>();
4      for (Map.Entry<Integer, List<ChangeBlock>> entry :
           variantIdToChangeBlocks.entrySet()) {
5        int variantId = entry.getKey();
6        // create changeVariant for every change process
             variant
7        ChangeVariant changeVariant = new
             ChangeVariant(variantId, original);
8        TaskTreeTemplate variantTTT;
9        List<ChangeOperation> changeOperations;
10
11       for (ChangeBlock changeBlock : entry.getValue()) {
12         // copy original TTT
13         variantTTT = createCopyOfTTT(agentId,
               roleAssignmentId, original);
14         // apply operations to copy
15         changeOperations = performChangesOnTTT(variantTTT,
               original, changeBlock);
16         // add changeOperations to changeVariant
17         changeVariant.addChangeOperations(changeOperations,
               variantTTT);
18       }
19       resultList.add(changeVariant);
20     }
21     return resultList;
22 }
```

Listing 6.2: Implementation of Change Variants Determination

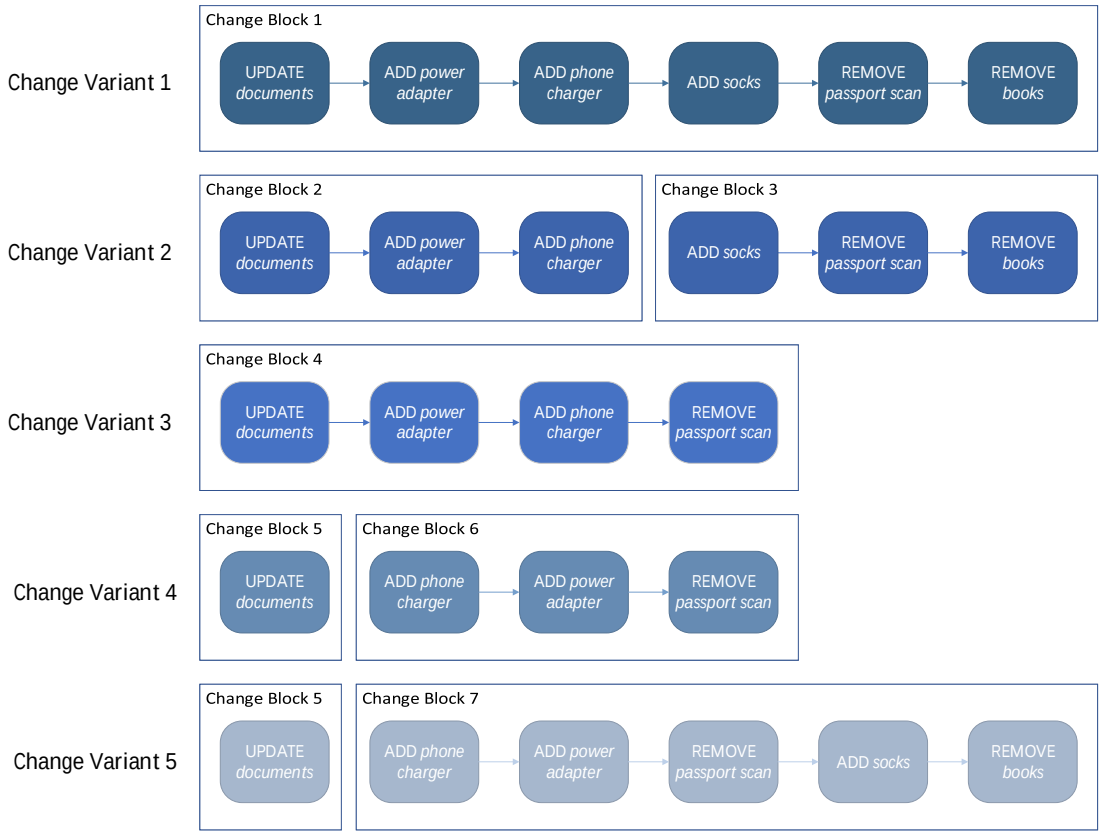


Figure 6.13: Change Variants determined for the Use Case of Section 3.3

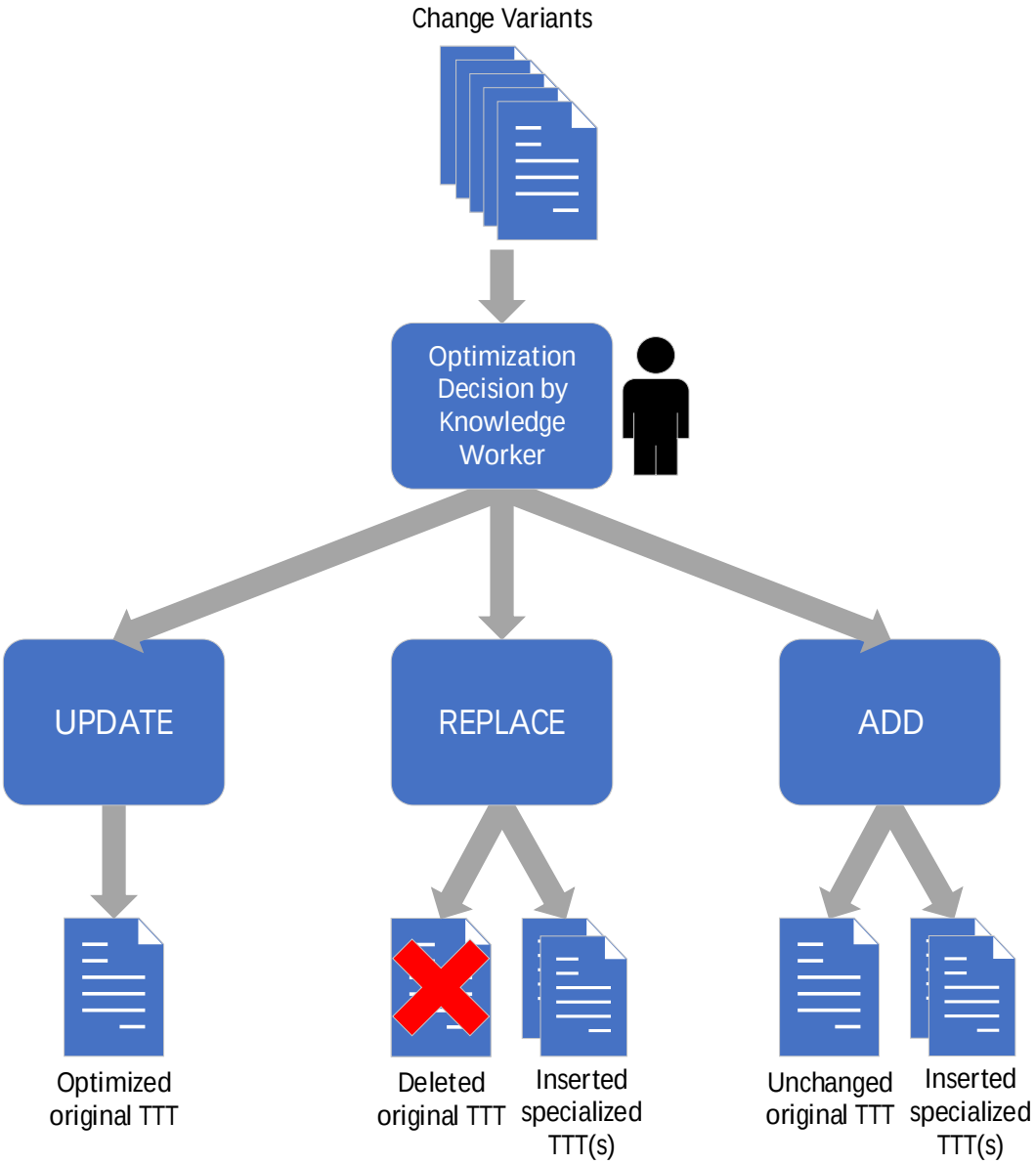


Figure 6.14: Knowledge Worker has three Optimization Options

6.2 Sub-Task List Template Generation

The *sub-task list template generation* extends the task list template evolution as it allows to identify nested inserted sub-trees within a TTI. While remove and update operations on task trees are well comparable, the comparison of insert operations is challenging since tasks (or task trees) can be inserted at different positions, with different parent elements and in different order. If tasks (or task trees) are inserted in nested form, the comparison of insert operations is increasingly difficult. To solve this problem, the sub-task list template generation is used when many nested insert operations have been applied to TTIs. It aims to identify the inserted sub-trees and to create a sub-tree from similar sub-trees that may be used to optimize an existing TTT.
The sub-task list template generation consists of several steps. In the first step, the change logs of the TTIs are checked for nested insert operations and add blocks are created. Each group of add blocks serves as input for the cluster mining algorithm (cf. Section 5.2) in the next step. The cluster mining algorithm then generates a TTT from the add blocks. Afterwards, a change variant (cf. Section 6.1.6) is created for each generated TTT. Subsequently, the task list template evolution is continued to analyze non-nested insert, remove and update operations. In the final step, knowledge workers may select one or more change variants to optimize the existing TTT.

Figure 6.15 shows a simple TTT that is instantiated at run time. Many tasks are added by knowledge workers while they collaborate in the KiP. As a result, sub-trees are formed under the TIs derived from the corresponding TTs of the TTT.

To identify these sub-trees, the previously cleaned change logs (cf. Section 6.1.1) of the corresponding TTIs are analyzed. Figure 6.16 illustrates the derivation of the add blocks from the change logs. Since only insert operations are included in the sub-task list template generation, only change log entries with the operation type *add* are considered. For each sub-tree identified in a change log, a so-called *add block* is created containing all insert operations of the corresponding sub-tree. An add block represents a sub-tree inserted under a particular parent element. In the next step, the add blocks from the different TTIs are grouped according to their parent element, i.e. the TT from the TTT on

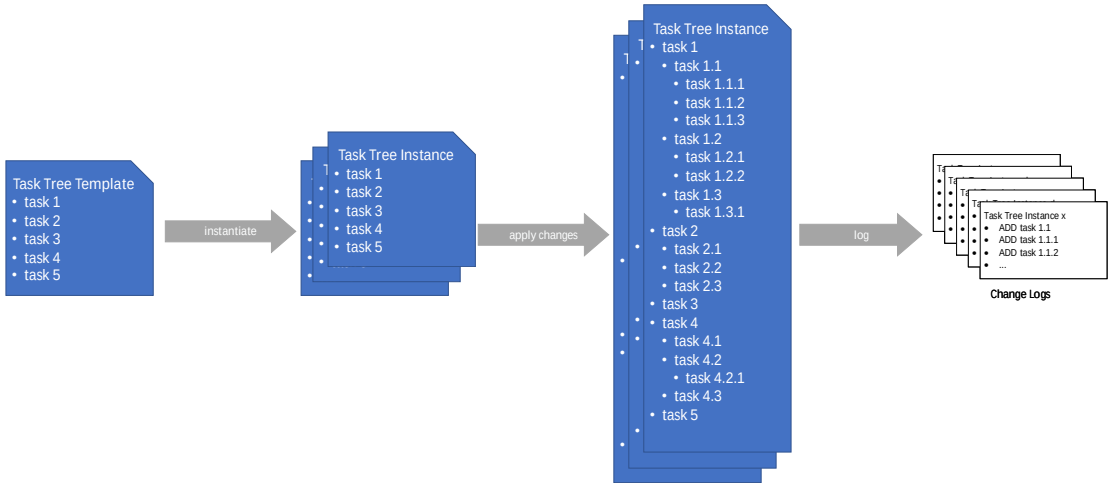


Figure 6.15: Multiple Add Operations on Task Tree Instances

which the TTIs are based (cf. Figure 6.17). The following steps are performed separately for all parent elements and their respective add blocks.

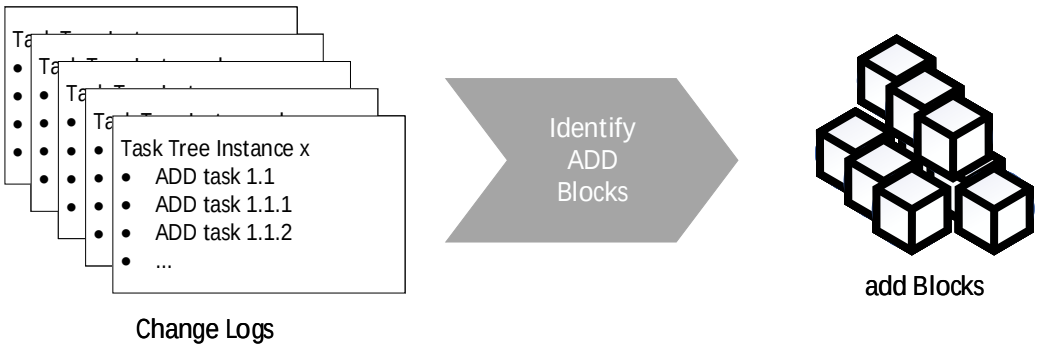


Figure 6.16: Identify Add Blocks

Once the different add blocks have been determined for all sub-trees, they must be converted into a format so that they can serve as input for the cluster mining algorithm. Therefore, in the next step, a TTI is created for each add block. Each TTI contains the tasks contained in the corresponding add block, i.e. the tasks that were inserted as a sub-tree during run time. The first child level inserted under the corresponding parent element represents the root level of the newly created TTI, i.e. the child elements are at task level 0. Figure 6.18 depicts the three derived add blocks and the corresponding TTI

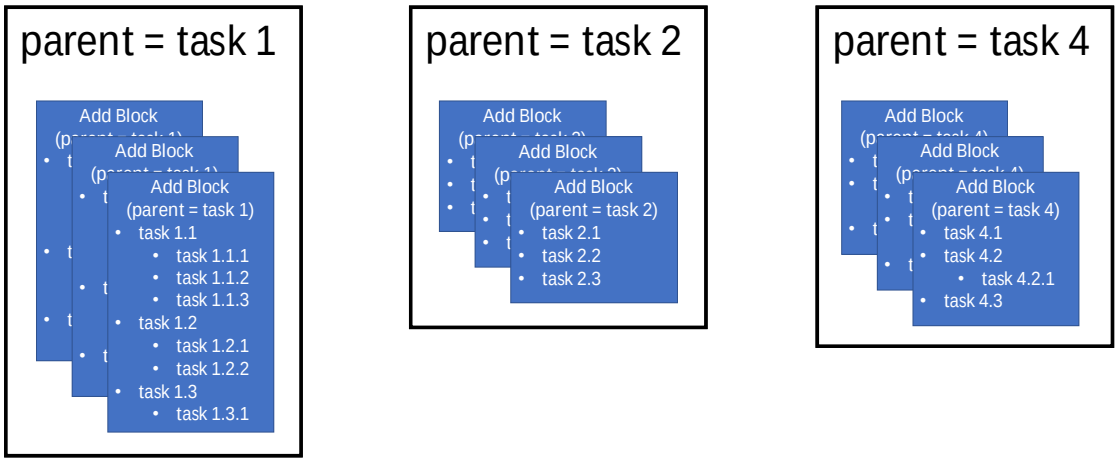


Figure 6.17: Grouping Add Blocks by their Parent Element

for the example in Figure 6.15.

When all add blocks have been transformed into TTIs, the cluster mining algorithm can be executed for each parent element and its TTIs. As a result, a TTT is generated for each parent element (cf. Figure 6.19). Subsequently, a separate change variant is created for each generated TTT. Therefore, a copy of the original TTT is created. The generated TTT is either directly added as a sub-template to the copy, or the tasks contained in the generated TTT are added individually to the copy.

In case a change variant was created, the used change log entries are removed from the change logs in the last step of the sub-task list template generation to prevent them from being included in the task list template evolution. Afterwards, the steps of the task list template evolution are continued to analyze the different non-nested insert, remove and update operations. At the end of the task list template evolution, in addition to the change variants derived by change mining, the knowledge workers may also select the generated change variants of the sub-task list template generation to optimize the corresponding template.

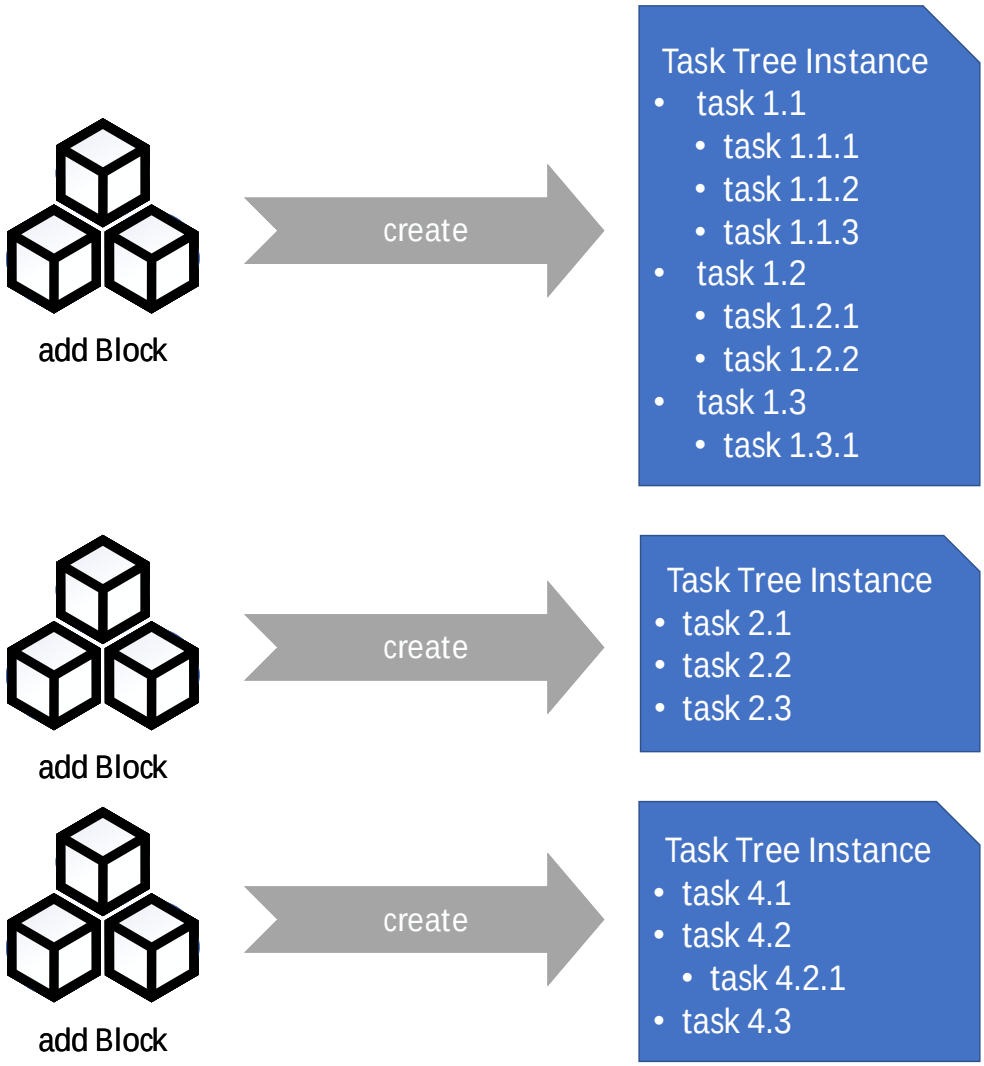


Figure 6.18: New Task Tree Instances are created for each Add Block

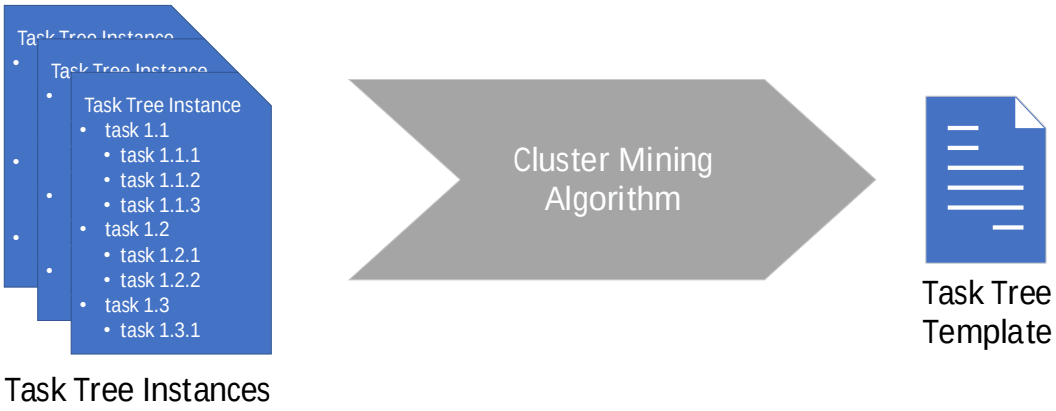


Figure 6.19: Sub-Task Tree Template Generation

6.3 Summary

This chapter introduces the task list template evolution pipeline for the continuous optimization of TTTs. First, the change logs of the TTIs are pre-processed. Subsequently, the change logs are checked for sub-trees and the sub-task list template generation is performed. In this way, inserted sub-trees are identified that may be used to optimize the respective TTTs. Afterwards, the multi-phase mining algorithm determines a causal net from the change logs. By evaluating the C-net, change variants representing the most frequent changes to TTIs are identified. Knowledge workers may select one or multiple change variants to optimize the TTT. As a result, changes made at run time may be included in the corresponding TTT or in new, specialized TTTs. If a similar KiP occurs in future, the optimized TTT may be instantiated by knowledge workers to support the KiP at run time. The goal of the task list template evolution is that knowledge workers do not have to adapt the TTIs derived from the optimized TTT as much as in earlier KiPs, since the most frequent changes from similar KiPs are already included in the optimized TTT. In this way, the workload of knowledge workers is considerably reduced and their support in the KiP is improved as they are provided with more suitable TTT.

In the course of this thesis, the change mining approach of the first proCollab prototype [3] was adapted to the new, advanced data model of the second proCollab prototype. In particular, the TTT optimization phase was adapted to the new data model. A TTT is

optimized by applying previously identified add blocks or change blocks, and thus their contained change operations, to the corresponding TTT.
The similarity analysis was integrated into the change mining approach as an enhancement to identify similar change operations. The similarity analysis is performed separately for each operation type, i.e. add, remove and update operations. This ensures that only change operations of the same type are grouped in a similarity group. The results of the similarity analysis, i.e. the similarity groups for add, remove and update operations, are used to determine the frequencies of similar change operations in order to be able to identify the most frequent change operations in a later step.
The sub-task list template generation extends the task list template evolution and allows nested insert operations to be analyzed and considered for optimization. Using only the task list template evolution, insert operations can be analyzed, but if they occur in nested form, their analysis is made considerably more difficult, since associated insert operations must be also taken into account. By using the sub-task list template generation the corresponding inserted tasks are automatically considered, because the cluster mining algorithm takes the different relations between the inserted tasks into account and consequently clusters the tasks with appropriate relations.
To enable knowledge workers to distinguish between the optimization of an existing TTT and the creation of an additional, specialized TTT, the concept of change variants was introduced. A change variant represents a copy of the original TTT to which a previously identified add block or change block is applied. For each add block or change block determined in the task list template evolution and the sub-task list template generation, a separate change variant is created. Knowledge workers make an optimization decision at the end of the task list template evolution by selecting one or more change variants to optimize an existing TTT or to create one or more new specialized TTTs.
By implementing the task list template evolution, the previously missing optimization phase in the second proCollab prototype was realized, i.e. the second proCollab prototype now supports the complete KiP lifecycle. The continuous improvement of TTTs ensures that knowledge and experience gained during a KiP is reused to better support knowledge workers in future KiPs.

However, there are possible improvements for the task list template evolution that may be considered for future work. Thus, the cluster mining algorithm could be used in addition to the multi-phase mining algorithm to derive the most frequent change operations from the completed TTIs. A subsequent comparison of the most frequent changes identified by both algorithms could give knowledge workers more opportunities to optimize TTTs.
In addition, it could be examined when an inserted sub-tree should be identified and analyzed using the sub-task list template generation and when using the task list template evolution. This could be determined on the basis of several threshold values, which would have to be adjusted accordingly for each use case. A threshold value could specify a minimum number of inserted tasks for a sub-tree and another one could define a minimum depth for nesting, i.e. how many child levels must be contained in the inserted sub-tree. If both thresholds were exceeded, the inserted sub-tree would be analyzed using the sub-task list template generation due to the deep nesting of many tasks. On the other hand, if both thresholds were not met, the task list template evolution would be used instead. For the other two cases, future research would have to investigate which alternative method would be more appropriate to use.

7

Conclusion and Outlook

7.1 Conclusion

The lack of system support for KiPs significantly hampers knowledge workers in achieving their goals. Therefore, knowledge workers today still widely employ simple paper-based task lists (e.g. checklists, to-do lists) or digital equivalents to plan and coordinate their work. For this reason, the proCollab research project aims to offer holistic, lifecycle-based support for KiPs in the form of task list management that enables knowledge workers to manage their task lists system-based.

The contribution of this thesis is the extension of the proCollab task management lifecycle by providing functionalities for the analysis of completed task lists. During the collaboration in a KiP, knowledge workers gain knowledge and experience shaping the knowledge-creating nature of KiPs. Hence, it is important to not disregard completed task lists, but to use them as a starting point for generating or improving a task list template. Due to the characteristics of a KiP, it is not always easily possible to provide knowledge workers with an appropriate task list template at run time. For this purpose, the task list template generation, which enables an automatic generation of task list templates from completed task list instances, was introduced and implemented in the proCollab prototype. Since completed task list instances contain knowledge and experience from previous KiPs, the task list template generation enables the creation of a common task list template based on knowledge that has contributed to the successful completion of a KiP and, thus, represents best practice. By automatically generating a task list template, the workload for knowledge workers at the beginning of a KiP can be significantly reduced, since they may decide whether they want to build a new task list template from scratch or select either an existing

task list template or the generated task list template for collaboration in the KiP.
In a KiP, task list instances like to-do lists are constant subject to change. Therefore, a continuous analysis of completed task list instances is necessary to improve existing task list templates on which the instances might be based on and to keep them up to date. To enable such a continuous evolution of task list templates, the task list template evolution was presented and implemented in the proCollab prototype. By analyzing the change logs of the task list instances derived from a task list template, frequent change operations are identified and used to optimize the corresponding task list template. Thereby, either the original task list template can be optimized or a new specialized task list template can be added. The use of frequently applied change operations for optimizing task list templates preserves the knowledge from previous KiPs in the optimized task list templates and significantly reduces the amount of redundant work knowledge workers have to perform as they do not need to optimize the task list templates themselves.
Many nested insert operations build sub-trees within a task list instance. To use the inserted sub-trees for the optimization of task list templates, the task list template evolution is extended by the sub-task list template generation. For this purpose, the added sub-trees are first identified and then a task list template is generated using the cluster mining algorithm. The new task list template is then used to optimize the existing task list template, either by inserting the generated task list template directly as a sub-template, or by adding the tasks of the generated task list template individually to the task list template to be optimized. The sub-task list template generation improves the task list template evolution, as the cluster mining algorithm takes all relations, including all hierarchical relations, between the tasks to be clustered into account and represents them correctly.
To realize the aforementioned concepts, the lifecycle services were implemented in the proCollab proof-of-concept prototype. The Lifecycle Service represents the centerpiece of the services and combines the functionalities of the subordinate services by providing the main methods for the task list template generation and evolution. The lifecycle services may be invoked via a central REST interface that calls the operations of the central Lifecycle Service. This provides a general interface for the various lifecycle services, which can be accessed by various applications (e.g. web client, mobile app, etc.).

To be able to test the various functionalities of the services, a test framework was developed and implemented providing several use cases for each concept.

7.2 Outlook

In future work, the different thresholds for similarity analysis and filtering of tasks should be investigated. The results obtained may be improved by optimizing these thresholds. For example, automatically adjusting the values based on the task list properties (homogeneous or heterogeneous lists) could provide improved results.
Furthermore, the similarity analysis should be improved so that it is able to better understand the semantic context. This could be achieved by using neural networks like the *word2vec* approach [22].
In addition to using the multi-phase mining algorithm to identify frequent change operations, the cluster mining algorithm could be used to generate a task list template from the completed, altered task list instances. A subsequent comparison of the two task list templates might give knowledge workers more options to improve the task list templates.
Another problem occurring with the optimization of task list templates is the so-called schema evolution [35]. If a task list template is modified by the task list template evolution, it should be checked whether the currently running task tree instances based on the corresponding task list template can be migrated to the new version of the template. If this is possible, all derived task list instances may be adapted in the same way as the corresponding task list template.

Bibliography

[1] Acampora, G., Vitiello, A., Di Stefano, B., van der Aalst, W.M.P., Gunther, C., Verbeek, E.: IEEE 1849: The XES Standard: The Second IEEE Standard Sponsored by IEEE Computational Intelligence Society. IEEE Computational Intelligence Magazine **12**(4), 4–8 (2017)

[2] Apprentissage & Optimisation Team (TAO): APRO. URL `https://github.com/lovro-i/apro`. (last accessed June 13, 2018)

[3] Beuter, F.: Design and Implementation of Task Management Lifecycle Concepts based on Process Mining. Master's thesis, Ulm University, Germany (2015)

[4] Cohen, W., Ravikumar, P., Fienberg, S.: A Comparison of String Distance Metrics for Name-Matching Tasks. In: Proceedings of the 2003 International Conference on Information Integration on the Web (IIWEB 2003), pp. 73–78. Acapulco, Mexico (2003)

[5] Damerau, F.J.: A Technique for Computer Detection and Correction of Spelling Errors. Communications of the ACM **7**(3), 171–176 (1964)

[6] van Dongen, B.F., van der Aalst, W.M.P.: Multi-phase Process Mining: Building Instance Graphs. In: Proceedings of the 23rd International Conference on Conceptual Modeling (ER 2004), pp. 362–376. Shanghai, China (2004)

[7] van Dongen, B.F., van der Aalst, W.M.P.: Multi-phase Process Mining: Aggregating Instance Graphs into EPCs and Petri Nets. In: Proceedings of the Second International Workshop on Applications of Petri Nets to Coordination, Workflow and Business Process Management (PCNWB 2005), pp. 35–58. Miami, USA (2005)

[8] van Dongen, B.F., de Medeiros, A.K.A., Verbeek, H.M.W., Weijters, A.J.M.M., van der Aalst, W.M.P.: The ProM Framework: A New Era in Process Mining Tool Support. In: Proceedings of the 26th International Conference on Application and Theory of Petri Nets (ICATPN 2005), pp. 444–454. Miami, USA (2005)

[9] Dumas, M., La Rosa, M., Mendling, J., Reijers, H.A.: Introduction to Business Process Management. In: Fundamentals of Business Process Management, pp. 1–31. Springer Verlag, Berlin Heidelberg, Germany (2013)

[10] Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. Ph.D. thesis, University of California, Irvine, USA (2000)

[11] Frey, B.J., Dueck, D.: Clustering by Passing Messages Between Data Points. Science **315**(5814), 972–976 (2007)

[12] Geiger, S.: Konzeption und Entwicklung einer auf Smartphones optimierten mobilen Anwendung für kollaboratives Checklisten-Management. Bachelor's thesis, Ulm University, Germany (2013)

[13] Gerber, M.: Design and Implementation of a Dynamic Web-based User Interface for the proCollab System Supporting Knowledge-intensive Business Processes. Master's thesis, Ulm University, Germany (2017)

[14] Gunther, C.W., Rinderle-Ma, S., Reichert, M., van der Aalst, W.M.P., Recker, J.: Using Process Mining to Learn from Process Changes in Evolutionary Systems. International Journal of Business Process Integration and Management **3**(1), 61–78 (2008)

[15] Haleby, J.: REST-assured. URL `http://rest-assured.io/`. (last accessed June 12, 2018)

[16] Klinkmüller, C., Weber, I., Mendling, J., Leopold, H., Ludwig, A.: Increasing Recall of Process Model Matching by Improved Activity Label Matching. In: Proceedings of the 11th International Conference on Business Process Management (BPM 2013), pp. 211–218. Beijing, China (2013)

[17] Köll, A.: Konzeption und Enwicklung einer auf Tablets optimierten mobilen Anwendung für kollaboratives Checklisten-Management. Bachelor's thesis, Ulm University, Germany (2013)

[18] Levenshtein, V.I.: Binary Codes Capable of Correcting Deletions, Insertions and Reversals. Soviet Physics Doklady **10**(8), 707–710 (1966)

[19] Li, C.: Mining Process Model Variants: Challenges, Techniques, Examples. Ph.D. thesis, University of Twente, The Netherlands (2010)

[20] MacQueen, J.: Some Methods for Classification and Analysis of Multivariate Observations. In: Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, pp. 281–297. Berkeley, USA (1967)

[21] Manning, C., Surdeanu, M., Bauer, J., Finkel, J., Bethard, S., McClosky, D.: The Stanford CoreNLP Natural Language Processing Toolkit. In: Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations (ACL 2014), pp. 55–60. Baltimore, USA (2014)

[22] Mikolov, T., Sutskever, I., Chen, K., Corrado, G.S., Dean, J.: Distributed Representations of Words and Phrases and their Compositionality. In: Proceedings of the 26th International Conference on Neural Information Processing Systems (NIPS 2013), pp. 3111–3119. Lake Tahoe, USA (2013)

[23] Miller, G.A.: WordNet: A Lexical Database for English. Communications of the ACM **38**(11), 39–41 (1995)

[24] Mundbrod, N., Beuter, F., Reichert, M.: Supporting Knowledge-intensive Processes Through Integrated Task Lifecycle Support. In: Proceedings of the 2015 IEEE 19th International Enterprise Distributed Object Computing Conference (EDOC 2015), pp. 19–28. Adelaide, Australia (2015)

[25] Mundbrod, N., Kolb, J., Reichert, M.: Towards a System Support of Collaborative Knowledge Work. In: Proceedings of the 1st International Workshop on Adaptive Case Management and Other Non-workflow Approaches to BPM (ACM 2012), pp. 31–42. Tallinn, Estonia (2012)

[26] Mundbrod, N., Reichert, M.: Object-Specific Role-Based Access Control. (Paper in Review). Ulm University, Germany (2018)

[27] Mundbrod, N., Reichert, M.: Process-Aware Task Management Support for Knowledge-Intensive Business Processes: Findings, Challenges, Requirements. In: Proceedings of the 2014 IEEE 18th International Enterprise Distributed Object Computing Conference Workshops and Demonstrations (EDOCW 2014), pp. 116–125. Ulm, Germany (2014)

[28] Mundbrod, N., Reichert, M.: Flexible Task Management Support for Knowledge-Intensive Processes. In: Proceedings of the 2017 IEEE 21st International Enterprise Distributed Object Computing Conference (EDOC 2017), pp. 95–102. Québec City, Canada (2017)

[29] Object Management Group: Business Process Model and Notation (BPMN), Version 2.0. URL `http://www.omg.org/spec/BPMN/2.0`. (last accessed May 15, 2018)

[30] Petri, C.A.: Kommunikation mit Automaten. Ph.D. thesis, University of Hamburg, Germany (1962)

[31] Pfiffner, M., Stadelmann, P.: Wissen wirksam machen: Wie Kopfarbeiter produktiv werden. Campus Verlag, Frankfurt am Main, Germany (2012)

[32] proCollab Team: Process-aware Support for Collaborative Knowledge Workers. URL `http://www.procollab.de`. (last accessed May 20, 2018)

[33] Pryss, R., Mundbrod, N., Langer, D., Reichert, M.: Supporting Medical Ward Rounds Through Mobile Task and Process Management. Information Systems and e-Business Management **13**(1), 107–146 (2015)

[34] Reich, D.: Konzeption und Entwicklung eines Cloud-basierten Persistenz-Systems für kollaboratives Checklisten-Management. Bachelor's thesis, Ulm University, Germany (2013)

[35] Rinderle, S.: Schema Evolution in Process Management Systems. Ph.D. thesis, Ulm University, Germany (2004)

[36] Scheer, A.W.: Business Process Engineering: Reference Models for Industrial Enterprises. Springer Verlag, Berlin Heidelberg, Germany (1994)

[37] Schubert, E., Koos, A., Emrich, T., Züfle, A., Schmid, K.A., Zimek, A.: A Framework for Clustering Uncertain Data. In: Proceedings of the 41st International Conference on Very Large Data Bases (VLDB 2015), pp. 1976–1979. Kohala Coast, Hawaii (2015)

[38] SCRUMstudy: Scrum Phases and Processes. URL `https://www.scrumstudy.com/whyscrum/scrum-phases-and-processes`. (last accessed June 14, 2018)

[39] Steinhaus, H.: Sur la division des corps matériels en parties. Bulletin de l'Académie polonaise des sciences **1**(4), 801–804 (1956)

[40] Thiel, N.: Konzeption und Entwicklung einer Web-Applikation für kollaboratives Checklisten-Management. Bachelor's thesis, Ulm University, Germany (2013)

[41] Ukkonen, E.: Approximate String-matching with Q-grams and Maximal Matches. Theoretical Computer Science **92**(1), 191–211 (1992)

[42] Vaculin, R., Hull, R., Heath, T., Cochran, C., Nigam, A., Sukaviriya, P.: Declarative business artifact centric modeling of decision and knowledge intensive business processes. In: Proceedings of the 2011 IEEE 15th International Enterprise Distributed Object Computing Conference (EDOC 2011), pp. 151–160. Helsinki, Finland (2011)

[43] van der Aalst, W.M.P.: The Application of Petri Nets to Workflow Management. Journal of Circuits, Systems and Computers **8**(1), 21–66 (1998)

[44] van der Aalst, W.M.P.: Process Mining: Discovery, Conformance and Enhancement of Business Processes. Springer Verlag, Berlin Heidelberg, Germany (2011)

[45] van der Aalst, W.M.P., Adriansyah, A., van Dongen, B.F.: Causal Nets: A Modeling Language Tailored Towards Process Discovery. In: Proceedings of the 22nd International Conference on Concurrency Theory (CONCUR 2011), pp. 28–42. Aachen, Germany (2011)

[46] van der Aalst, W.M.P., Rubin, V., Verbeek, H., van Dongen, B.F., Kindler, E., Günther, C.W.: Process Mining: A Two-Step Approach to Balance Between Underfitting and Overfitting. Software & Systems Modeling **9**(1), 87–111 (2010)

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

[48] Weske, M.: Business Process Management: Concepts, Languages, Architectures. Springer Verlag, Berlin Heidelberg, Germany (2012)

[49] Ziegler, J.: Konzeption und Entwicklung eines Cloud-basierten Servers für kollaboratives Checklisten-Management. Bachelor's thesis, Ulm University, Germany (2013)

List of Figures

List of Tables

List of Listings

Name: Simon Stöferle Matriculation number: 754157

Statutory Declaration

I hereby affirm that I wrote this thesis independently and that I did not use any other sources or tools than the ones specified.

Ulm, June 15, 2018 .....

Simon Stöferle