ulm university universität
uulm

Universität Ulm | 89069 Ulm | Germany

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

Supporting Task Constraints and Dependencies in Knowledge-intensive Processes

Master's thesis at Ulm University

Submitted by:
André Lang
andre.lang.ger@gmail.com

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

Supervisor:
Nicolas Mundbrod

2018

Version from August 7, 2018

Composition: PDF-LaTeX 2_ε

Abstract

Knowledge-intensive processes are widely found in modern societies and important to many fields of work. Driven by knowledge gained only during their execution, this type of process brings along challenges like a gradually emerging structure, uncertainty and frequent changes. Through contribution and communication the knowledge workers involved in the process shape and improve it while advancing towards a common goal. One very important part of the knowledge are the dependencies that naturally exist between the tasks on which these workers perform. With the latter often being spatially divided, expressing the dependencies existing for their jobs is a determining factor for success. Relying on insufficient software systems or, even worse, on paper-based solutions for coordination proves to be error-prone and lacks reliance in practical scenarios. Adequate aid in form of information systems, specifically designed for knowledge-intensive processes and their accompanying challenges, and operated on by the knowledge workers themselves, is needed. The work on such systems is a still ongoing endeavor and in need of concepts and solutions.

This work presents a concept for the support of constraints to express task dependencies in knowledge-intensive processes. With the focus on the unique challenges coming with the latter, the concept puts great emphasis on providing guidance instead of strict ruling, adaptability to frequent changes and usability in practical scenarios. A rule-based, declarative approach is laid out for applying the concept, designed to be ready for extension, to various systems alongside other, already existing functionality. Based on it, a catalog of constraints is given, with clear semantic meanings and effects, tailored towards the use in knowledge-intensive processes. A proof-of-concept prototype for it was implemented for the process-aware Support for Collaborative Knowledge Workers (proCollab) system, an example of an adequate and sophisticated solution.

Acknowledgment

I wish to express my sincere appreciation to my supervisor Nicolas Mundbrod for offering ample advice and constructive criticism, helping me to form and shape this work in theory and practice. He clearly showed his expertise and knowledge for the time of my writing and even before and provided me with invaluable support. With the proCollab system being the child of his research and efforts I thank him for inviting me to join in and for helping me find a way through it's great expanse of capabilities.

I would like to thank Khaled Sherif for his advice, motivation and aid. It was always encouraging and inspiring to talk to him and exchange ideas.

My special thanks go to my parents for supporting me and to my sister for supplying me with bags of roasted beans allowing me to put myself in a heightened state of caffeinated consciousness when the lights went dim and the work was not quite done, yet.

Contents

1

Introduction

A shift in developed countries seen in the last decades led to the rise of processes driven mainly by the knowledge of their participants [1]. These Knowledge-intensive Processes (KiPs) like, e.g., software development projects, automotive E/E development or complex financial services are naturally predisposed for frequent changes and accompanying uncertainty as knowledge grows during their execution [2] [3]. The ability to share information and the collaboration between the knowledge workers becomes a major factor towards success and the need for sophisticated information systems that adequately attend to the unique characteristics of KiPs is high. They have to provide means to support the collaboration and sharing of knowledge between knowledge workers, while the process gradually emerges and frequently changes.

A very important part of this knowledge are the dependencies between work-items and proper support when handling them. Many jobs have to be done adhering to the relationships in between them and while relying on the outcome of other individual's assignments. Supporting systems therefore need to also provide options for manifesting these dependencies so that workers can perform their jobs while adhering to them.

The term constraint in the context of this work is defined with KiPs and information systems that support these in mind:

Definition 1 (Constraint)**:** *Within the context of this work, a constraint describes the dependencies between stateful, task-centric work-items and provides guidance for performing work on them by proposing restraints on changing their states and, where needed, the states of their context(s) and their sub-items. The proposed restraints are set up and revoked based on the states of work-items connected through various dependencies or relationships.*

1.1 Problem Statement

Even when not explicitly stated in the system or on a task list, dependencies like e.g. a relationship between tasks or induced activities do very well exist. Thus they have to be included in the representation of the process and it's items. Making people aware of dependencies is an essential part of task-centric process support. It allows the individuals to perform while adhering to the dependencies or basing their decisions on the dependencies' implications. The dependencies and their effects get modeled through constraints, added to the process and the information system. Successfully supporting these constraints in a knowledge-driven environment is not trivial and several unique challenges and problems have to be faced.

To concrete the topic of this work, two of its most important terms are defined in the following:

KiPs Knowledge-intensive processes in this work are defined analogous to knowledge-intensive business processes: "Knowledge-intensive processes (KiBPs) are processes whose conduct and execution are heavily dependent on knowledge workers performing various interconnected knowledge intensive decision making tasks. KiBPs are genuinely knowledge, information and data centric and require substantial flexibility at design- and run-time." [4].

Dependency A dependency refers to a relationship between work-items such that one cannot reach a certain state until one or more other work-items have reached specific states..

The characteristics of KiPs to be considered can be summed up as below [2].

Uncertainty Complexity arises through a high number of influencing factors. The process' structure and execution cannot be predetermined. This uncertainty also covers the sequence and dependencies between tasks or activities.

Goal Orientation In order to help with individual goal-setting, sub-goals can be derived from common goals. These possibly newly added or refined tasks can be modified or removed eagerly during the run-time of the process.

Emergence The constant evaluation of influencing factors does lead to adaptions occurring frequently. Thus, KiPs emerge gradually through agile notion and incremental phases.

Growing Knowledge Base The increasing amount of knowledge has to be made available to collaborating knowledge workers and departments. Resulting work-items like, e.g., tasks shape the process step-by-step while increasing complexity and dependencies.

Through these characteristics, a number of problems originates that have to be taken into consideration in order to successfully support constraints:

Problem 1 Knowledge workers themselves do create constraints with usability conceivably threatened by interplays and comprehensibility of the constraint's meanings and effects.

Problem 2 Frequent changes and adjustments call for high adaptability.

Problem 3 Constraints have to work on task-centric work-items acting based on their current states.

Problem 4 In addition, Constraints have to consider the hierarchical and contextual organization of the work-items.

Problem 5 Users might need to overrule the effects of constraints thus favoring guidance over strict ruling.

Problem 6 A number of regularly needed constraint types has to be identified and adopted for the use with KiPs.

Problem 1: Constraints Are Not Created and Used by BPM Experts In the case of KiPs, refinements to the process are done by various knowledge workers acting on it at run-time [5]. The individuals working on the KiPs are also responsible for creating the constraints [1], with the need to do so arising as they work on the process and gain new knowledge. Because more constraints bring increased complexity and often hard to grip interplay user acceptance can become endangered. One constraint might hinder the

solving of another one - a consequence that might not have been clear to the different creators when they set the constraints up. Usability therefore is of high importance. This means having semantically clearly defined constraints with comprehensible effects that can be easily grasped by workers coming from different fields.

Problem 2: Frequent Changes and Adaptations With the structure of a process evolving over time and it's later manifestations barely known earlier on, trying to lay out even transient portrayals of a KiPs is hard to achieve and realistically rarely possible. As an example, the diagnostic-therapeutic cycle that can often be seen in the medical field is shown in Figure 1.1. A top level overview of a medical patient treatment process, it's numerous sub-steps and elements do only emerge during it's iterative execution. Even with a more detailed structure laid out during later stages, improvements and refinements are still being added during the process' execution. This nature of being gradually emerging leads to the common inability to predefine constraints. Instead, constraints will be extensively added or removed at run-time of the process. This is in opposite to structured process types, where constraints are often used to predefine sequences of activities during planning and design-time. Further changes are done to the work-items, as, e.g., existing ones get moved into different task-lists or new ones are added to the numbers of items a constraint is already putting it's effect on. Constraints therefore need to be aware of these changes and be provided with means to readily adapt to them.

Problem 3: Constraints Have to Focus on the Various States of Task-Centric Work Items For precisely monitoring the progress, work-items like tasks and similar entities can have multiple states available to them and go through them while work is being performed. By choosing and, again, refining states, users denote the status of the work. Constraints monitor the work-items and base their effects on the current states of relevant ones, possibly limiting or prohibiting state changes when and where needed.

Problem 4: Constraints Have to Work Considering Hierarchy and Context Tasks and task-centric work items like task lists are used for organizing and representing

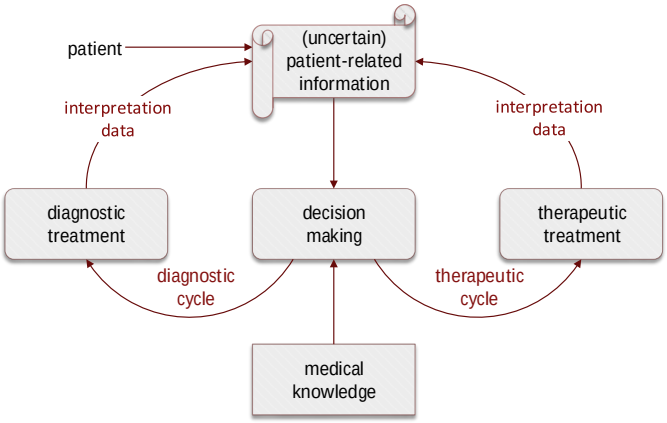


Figure 1.1: Diagnostic-therapeutic cycle, based on [6].

assignments and help with the goal orientation of the involved individuals. Based on the work-items, the jobs and the work-to-do are assigned to specific knowledge workers. To help with that, tasks and similar entities of a system can be organized in a hierarchical structure and by context. E.g. in the process-aware Support for Collaborative Knowledge Workers (proCollab) system tasks can be grouped together in task lists. Task lists then again can be grouped under a process. This creates a hierarchy. Constraints need to recognize this structure and span their effects accordingly and with respect to various corresponding settings made available to users.

In this work, the term context, when related to work-items and hierarchy, is used as a synonym for task lists or (sub-) processes. Context can therefore be defined as follows:

Definition 2 (Context)**:** *Area of application for constraints and dependencies formed by a CSE. Covers the CSE and all of its stateful sub-entities.*

Problem 5: Constraints Have to Provide Guidance Instead of Strict Ruling As mentioned above, the complexity coming with constraints in form of dependencies and consequences they introduce is not always easily recognizable by users in advance. Furthermore, situations can arise that call for alterations or exceptions to what is currently called for by existing tasks and constraints. In case of problems or when a sudden request for change arises, adaptiveness in the form of an option to overrule or violate constraints is required. With gradually emerging KiPs the aim of constraints is not to predefine strict

rules for, e.g., automating process execution, but to provide adjustable guidance to users during run-time.

Problem 6: Regularly Needed Constraint Types Have to Be Identified In addition to the problems originating from KiPs a more general one is to identify which types of constraints an information system should support. As a great number of different dependencies for task-centric entities can be imagined, frequently needed ones that can prove to be helpful in many various use cases have to identified. Noteworthy is that examples for constraints that are found in various literature often refer to cases of non-KiPs scenarios and therefore cannot be adopted one-to-one. New constraints have to be designed specifically for their use in KiPs based on these examples.

1.2 Contribution

The foremost goal of supporting constraints for KiPs is to complement the existing functionality of an information system by introducing means to express natural dependencies and relationships for work-items. Constraints propose restrictions to state changes based on these work-items' dependencies and current states. This work lies out a concept for the support of constraints in KiPs considering the specific problems coming with the type of process at hand. The concept is created to be readily extended with further functionality and to be easily adopted for various systems. A prototype implementation for the proCollab system serves as a proof of concept and to provide examples for how to turn the theory into practice.

This works contribution can be summed up as follows:

- A concept for the support of task constraints and dependencies in KiPs is presented.
- Frequently required and useful constraints were identified for the topic at hand and their semantic meaning and required logic worked out.

- Constraints are designed to perform on stateful, task-centric work-items while recognizing their hierarchical and contextual organization.
- Constraints are meant as tools for providing guidance to users instead of imposing strict ruling.
- A generic and comprehensible concept for supporting the different constraint types is laid out.
- Concept based on a rule-driven, declarative approach utilizing the Drools rule engine.
- Concept designed to be applicable to various information systems.
- A proof-of-concept prototype was implemented for the proCollab system.

Note that the subject of this work was oriented solely on the back-end of task management systems. The inclusion of the front-end and user-interface-related functionality for supporting constraints was not part of this work.

First, a number of desirable constraints and types of constraints was identified and selected. The decision for the selections were based on which types would provide an information system with a baseline for often found and required dependencies. The concepts for different categories or types of constraints are described in a catalog of constraints. Being wary of offering the possibility to increase their numbers with future additions, a number of examples for other types is provided as well.

Furthermore, all constraints were designed to follow a common and well to comprehend approach. The meaning of each constraint and their effects facilitates the understanding of knowledge workers coming from various fields. This relates to *Problem 1: Constraints Are Not Created and Used by BPM Experts* (see Section 1.1) and aims at improving the usability and user acceptance.

With the foremost challenge of KiPs being uncertainty and frequent changes, the concept emphasizes adaptability to such. The shown solutions and utilized technology make it easy to react to adjustments and alterations. This focus on solving *Problem 2: Frequent Changes and Adoptions* (see Section 1.1) was a focal point of this work. Several possible future additions aiming at further enhancing the adaptability are presented.

Instead of automating process execution the goal is to enrich task management systems focusing on stateful, task-centric entities. First, this means having constraints monitor the states of the entities and reacting upon reaching specific ones (*Problem 3: Constraints Have to Focus on the Various States of Task-Centric Work Items*, see Section 1.1). Second, the concept focuses on working with entities being organized in a hierarchical structure and grouped in various contexts (*Problem 4: Constraints Have to Work Considering Hierarchy and Context*, see Section 1.1). Constraints do recognize the hierarchy and context of the work-items relevant to them and spread their effects accordingly and as configured.

It was very important to design all constraints and their effects so they offer guidance through proposals instead of imposing strict ruling (*Problem 5: Constraints Have to Provide Guidance Instead of Strict Ruling*, see Section 1.1). A system employing this concept is left to decide on how exactly to react to the restraints proposed by constraints. This allows it to run as flexible as needed in practical scenarios where decisions might change and the ultimate choice might not be put on pure automation but rather on the knowledge carried by human factors.

A rule engine (Drools Expert by Red Hat/JBoss) was chosen as the driving technological component for the constraints and functionality. Instead of having nested logical conditions in imperative code, this allows for a rule-driven, declarative and often autonomous approach. Logic is encapsulated in compact, fine-grained rules and the utilization of facts on the part of the rule engine naturally fosters the reaction to changes. In addition to realizing the constraints, this work also lies out examples on how to employ the rule engine for supporting minor work-flow related actions.

Overall, the concept is designed to be easily adaptable to different systems and their entities. The concept's implementation is to be added and work alongside of existing functionality. As a proof of concept, a prototype implementation for the selected constraints and their basic functionality was done for the proCollab system. This system also provides an example of a highly sophisticated tool for adequately supporting KiPs.

1.3 Structure of the Thesis

First, the fundamentals and related work are presented in Chapter 2. After that, the concept together with it's constraints will be discussed in Chapter 3. Next, Chapter 4 grants some insight into the implementation of the prototype and the rules written for one of the constraints. And finally, Chapter 5 sums up the content and results of this work and provides an outlook of what could be added in the future.

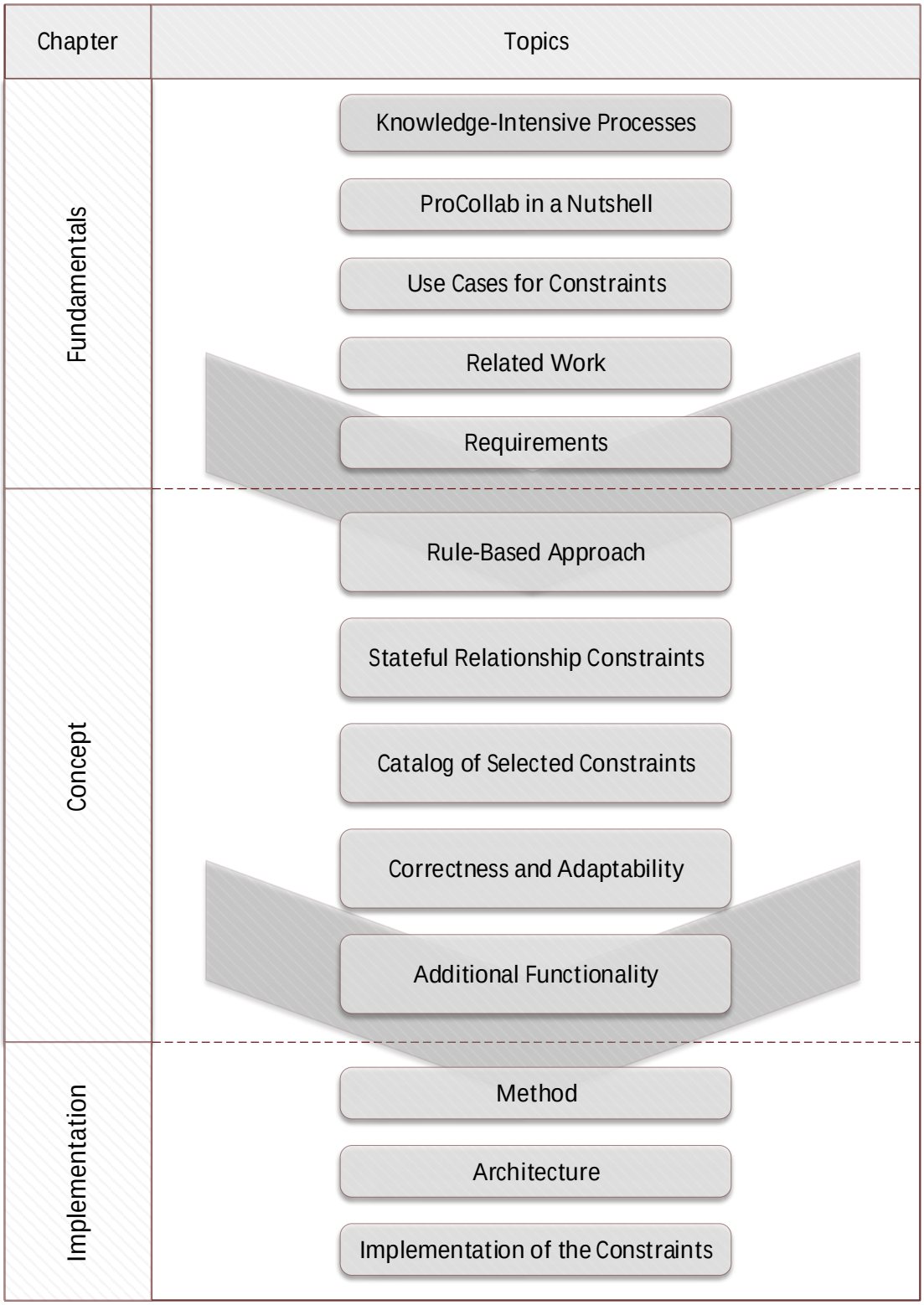


Figure 1.2: Roadmap of this work's main chapters.

2

Fundamentals

The core of the work at hand are the constraints and KiPs. This chapter focuses on the latter, describing the subject, challenges and work related to the KiPs including the proCollab system. At first, in Section 2.1, the characteristics and challenges of KiPs are explained. After that, the proCollab system is briefly introduced in Section 2.2 as a way to support KiPs, and its state management explained in Section 2.2.2. Next, use cases and practical examples for constraints and dependencies are given in Section 2.3. An overview of influential and related work can be found in the succeeding Section 2.4. Concluding this chapter, the requirements for the support of constraints in KiPs, derived from the points before, are laid out in Section 2.5.

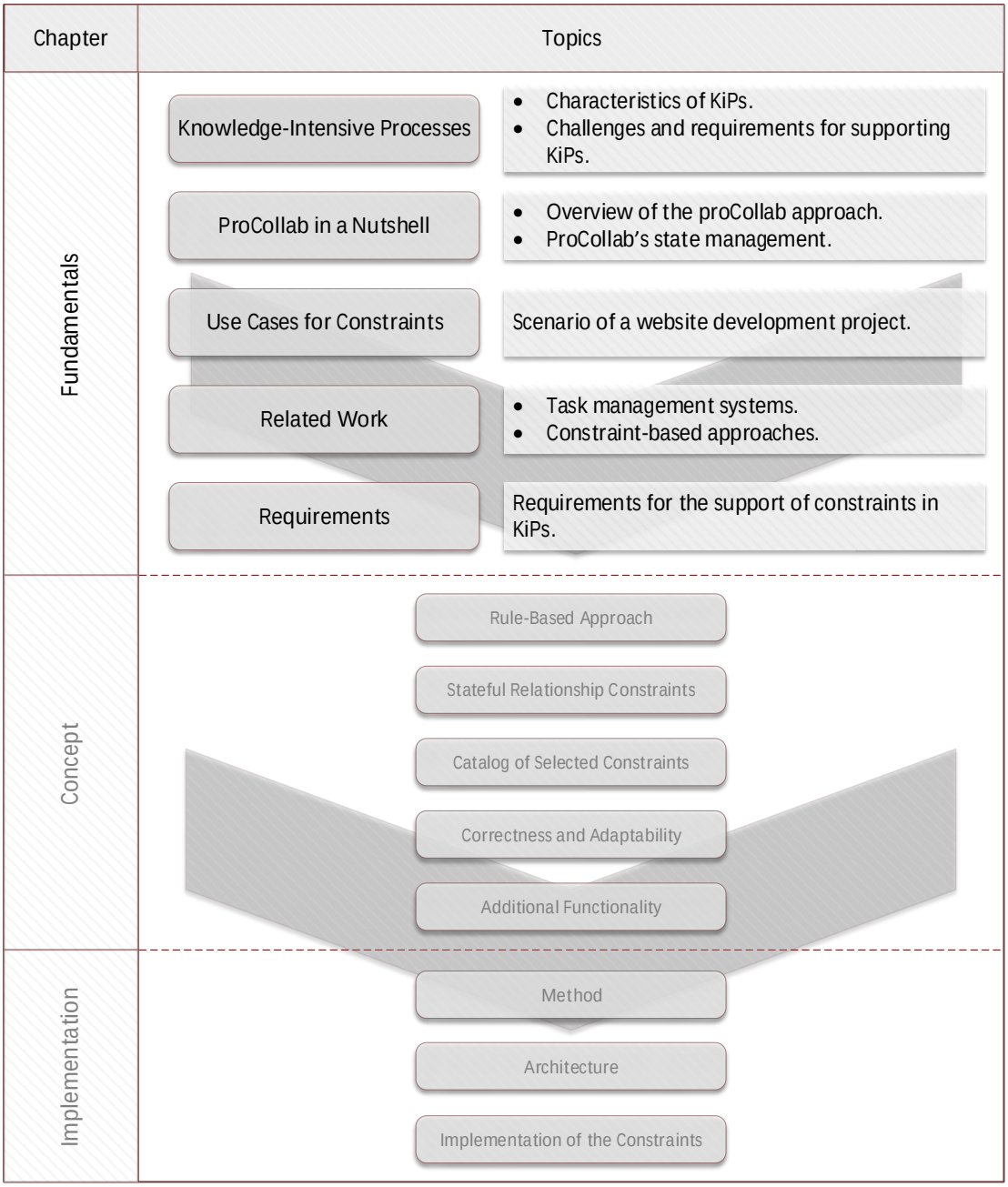


Figure 2.1: Roadmap highlighting the chapter on the fundamentals.

2.1 Knowledge-Intensive Processes

A shift towards knowledge work (KW) in developed countries makes KiPs become more and more important [2], [7]. Knowledge workers provide their expertise while iteratively gaining an increasing amount of collaboratively shared knowledge. KiPs are more influenced by iteratively gained information and knowledge than structured processes. They show an increased amount of uncertainties and unknown interdependencies between activities. By this, they differentiate from other, more structured types of processes and bring their very own challenges to face.

2.1.1 Terms and Definitions

The very base of the work performed on KiPs is the knowledge itself. It sums up information learned by the workers which can be applied to the tasks ahead and is communicated and shared with others [2].

Definition 3: *"[...] is a fluid mix of framed experiences, values, contextual information, and expert insights that provides a framework for evaluating and incorporating new experiences and information. It originates and is applied in the minds of knowers. In organizations, it often becomes embedded, not only in documents or repositories, but also in organizational routines, processes, practices, and norms." [8].*

Knowledge is generated and/or further expanded by performing knowledge work - explained in [2], [9] as:

Definition 4: *"[...] is comprised of objectifying intellectual activities, addressing novel and complex processes and (work) results, which require external means of control and a dual field of action." [9].*

Individuals performing this knowledge work are referenced as knowledge workers, a term defined by [10] as follows:

Definition 5: *"[...] have high degrees of expertise, education, or experience, and the primary purpose of their jobs involves the process and accomplishment of knowledge work." [2].*

2.1.2 Characteristics of Knowledge-intensive Processes

As stated before, the main characteristics when performing KiPs as laid out by [2] can be seen in their uncertainty, goal orientation, emergence and growing knowledge base. Figure 2.2 lists the range of various process types going from clearly structured processes on the left to the more uncertain and continuously evolving KiPs on the right. Following that same direction, the shift goes from routine and technology towards the involved people and the need for coming up with creative solutions and adaptations.

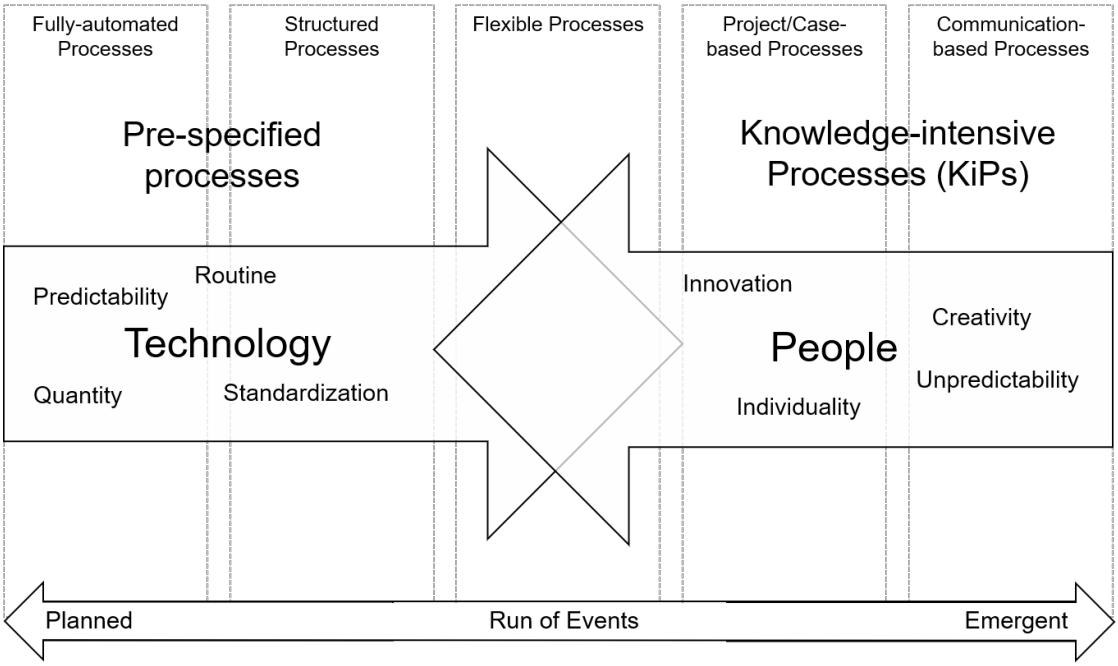


Figure 2.2: Types of processes, reprinted from [11].

As knowledge workers are performing their work and generate new knowledge and insights, changes and additions come frequently. This character of emergence is typical to KiPs more so than for processes closer to the opposing end, with the latter often being easier to plan in advance. Orientation for the knowledge workers is provided in form of common goals that are linked to each process, phase and sub-phase. These goals are important to define and share so that the individuals' efforts can be steered towards success.

The resulting new knowledge artifacts and a knowledge base that keeps growing during the process' execution form the baseline for the ongoing work as well as for future projects. Much of the information and knowledge generated shows interdependencies, as workers of often different departments need to access and utilize it. Storing this knowledge and artifacts while making them accessible during the collaborative endeavor is crucial to the KiPs' progress and success.

Figure 2.3 depicts the process when performing knowledge work. Note that each step can be skipped or repeated as needed. An individual deducts an assignment from provided information and goes through a sequence of orientation, planning, action and evaluation. The possible result is the adaption of the initial plan and process to the new findings. While some of the steps (2, 3, 6 and 7) are performed as actual, practical activities, the referential field of action states others as work of theory (4, 5 and 8). This theoretical work needs to be saved and formalized as tasks or through documentation. Without this, it would not be preservable or accessible by other workers. The evaluation in step 8 can lead to an alteration of the original plan, highlighting how changes appear due to an individuals gained insights. In summary, each individual iteratively creates practical results and theoretical knowledge that naturally and regularly lead to adaptations for the process.

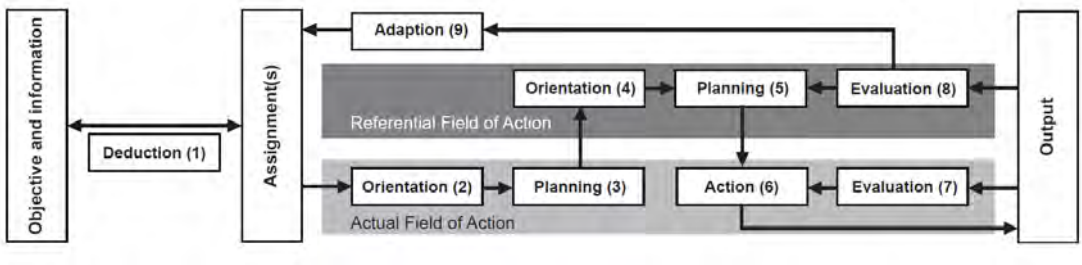


Figure 2.3: The process of performing knowledge work, reprinted from [2].

2.1.3 General Challenges and Requirements

A selection of the challenges when supporting KiPs as stated in [3], focusing on the most important aspects for this work, is given below:

Meta-model design At first, it has to be built upon a meta model which is able to provide agile support for knowledge workers while respecting the characteristics of KiPs. An important aspect is the use of task-centric work items like tasks, task lists and checklist (see "Task trees" in Section 2.2.1).

Lifecycle support Furthermore, the system has to allow to correctly reflect, monitor and manage the complex lifecycle of the KiPs. By this, it offers ongoing guidance to the involved knowledge workers.

Variability support Users would work with templates to plan ahead and aid with their assignments. Knowledge workers would refine tasks into sub-tasks in order to have more compact work-items at hand. Alterations to the templates and creation of new ones therefore is likely frequent and such the requirement for supporting a high amount of variability arises. In addition to this, there has to be a way to help knowledge workers find the correct templates for their needs.

Context support Lastly, the high number of knowledge workers can be seen working in various contexts and often in more than one context at the same time. Successful collaboration between these individuals depends on representing their up-to-date roles, abilities and affiliation.

In view of this insight, a number of requirements for systems supporting KiPs was derived [3]. Those shown below are just a few out of many and represent some of those more closely related to the topic of this work.

Task-centric entities The baseline for the knowledge workers' work is formed by task-centric entities. These work-items are used by the knowledge workers for planning as well as for performing their work. A clear structure and sufficient amount of attachable information like, e.g., task goals, descriptions, authors and priorities has to provided.

Task Dependencies Furthermore, hierarchical and temporal dependencies for tasks allow to set up a clear structure for the tasks' execution. This also forms the baseline for the to-do lists and checklists which are utilized for planning, performing and for quality assurance. In addition, being able to put constraints on the execution of tasks allows to express relationships.

Adaptive task structures Another important requirement is a task structure that is adaptive enough to support creating, altering and removing tasks during run-time and also to transform a task's structure into another sound and reasonable one.

Context rules "[...] knowledge workers should be enabled to deposit contextual rules in proCollab. Based on the current context of knowledge workers, the latter may be leveraged to adjust the provided guidance provided by a CI. Therefore, proCollab need to allow knowledge workers to define context rules for CTs and to enable knowledge workers to change these rules on demand during run time." ([3]) (Note: The abbreviations "CT" and "CI" refer to collaboration templates and collaboration instances, respectively.)

The following Table 2.1 gives a summarized overview over all challenges and their related requirements when supporting KiPs. For more insight and a detailed elaboration on the stated and additional challenges, findings and requirements please consider [3]. All identifiers found in the table are equivalent to those of the source.

ID	Challenge Name	Related Requirements
C1	Meta-Model Design	Task-centric entities (R1), Task Dependencies (R2), Adaptive Task Structures (R3), Meta-Model Comprehensibility (R4), Meta-model Extensibility (R5)
C2	Lifecycle Support	Lifecycle Entities (R6), Collaboration Instance Archiving (R7), Collaboration Schema Evolution (R8), Collaboration Instance Generalization (R9), Run-time Recommendations (R10), Version Control (R11)
C3	Variability Support	Collaboration Template Families (R12), Collaboration Template Guidance (R13), Collaboration Template Configuration (R14), Collaboration Template Annotation (R15)
C4	Context Support	Context Model (R16), Context Rules (R17)
C5	View Support	Personal Views (R18), Context-specific Views (R19)
C6	Authorization Support	Authorization Model (R20)
C7	Synchronization Support	User Encouragement (R21), Mobility (R22), Events (R23)
C8	Integration Support	Application Programming Interface (R24), Process Integration (R25)

Table 2.1: Challenges and requirements for KiPs, based on [3].

2.2 ProCollab in a Nutshell

As KiPs come with unique characteristics, an information system needs to have its functionality specifically geared towards the demands of this type of process. The following sections explain how the proCollab system helps to adequately support KiPs (see Section 2.2.1 and Section 2.2.2).

2.2.1 The proCollab Approach

The proCollab approach follows a generic, open and agile concept which makes use of instantiate-able templates for processes, task lists and tasks [1]. Each of these entities is *stateful*, meaning it is provided with a current state so that the status of the work on the KiPs can be monitored in detail and at any given time.

A compact and accurate summary of the proCollab approach is:
"The proCollab (process-aware Support for Collaborative Knowledge Workers) approach was developed to establish a generic, lightweight and lifecycle-based task management support for KiPs. In this context, the approach is capable to provide customizable coordination support for a wide range of KiPs in the shape of projects, cases or investigations. Further, all kind of prospective (to-do lists) and retrospective task lists (checklists) are supported to enable better task coordination and synchronization within the knowledge-intensive processes. [...]
As a foundation, proCollab employs processes, task trees and tasks as its key components. To establish the required lifecycle-based support, knowledge workers may define process, task tree, and task templates in proCollab. These templates enable the definition of best practice for planning and coordination as well as the preservation of existing process knowledge. At run time, knowledge workers may instantiate specified templates or create process, task tree, and task instances from scratch." [12]

The proCollab system is the prototype for the approach. It is composed of a server-sided backend application and a web-based user client.

Figure 2.4 depicts the main aspects of working with this system. Users mainly work on a graphical presentation of task lists and tasks. Several templates are available for instantiation and, based on the logs and records of previously performed work, these templates get refined and optimized.

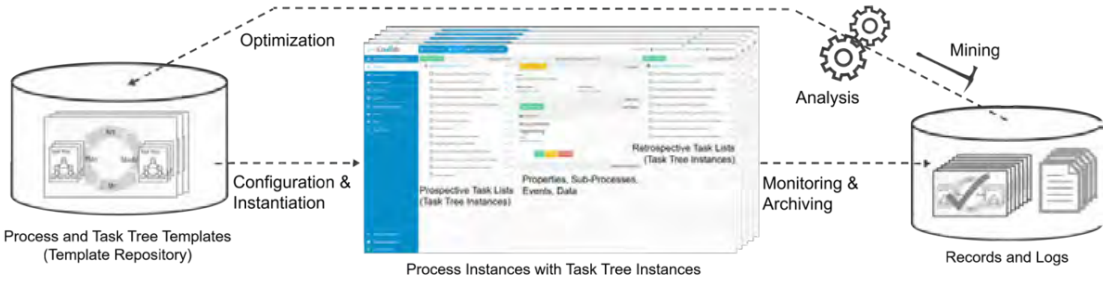


Figure 2.4: ProCollab approach and user interface, reprinted from [12].

Figure 2.5 shows the major components of the proCollab system. Based on [1] these can be described as follows.

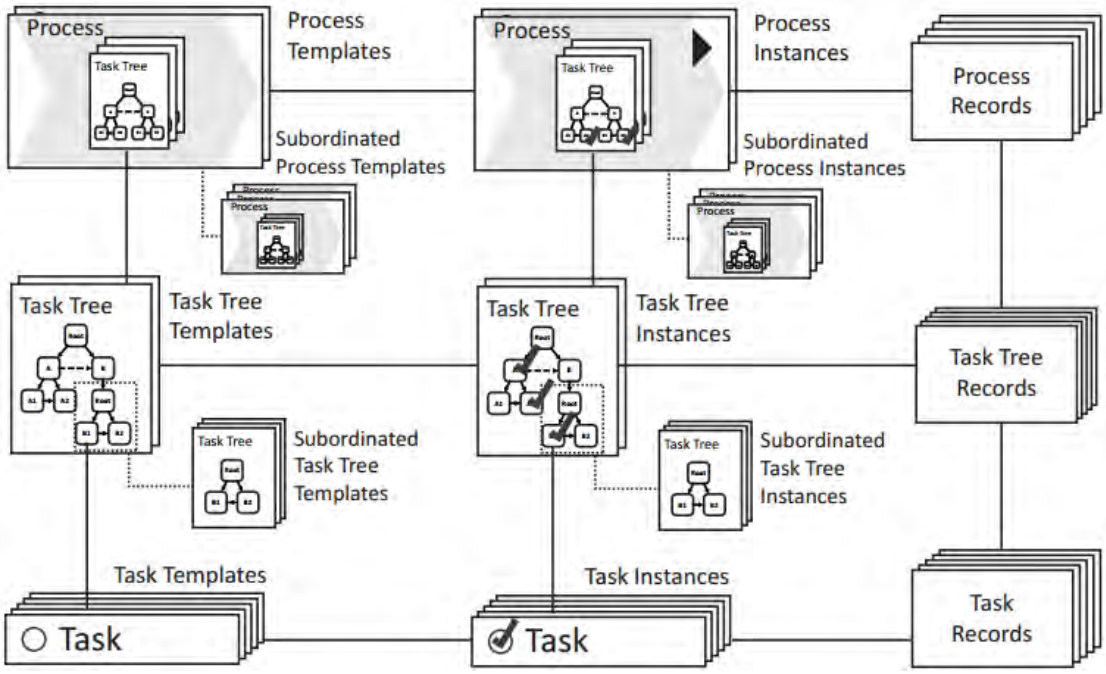


Figure 2.5: Overview of the proCollab system's components, reprinted from [1].

Processes expose an end goal that knowledge workers want to achieve. In addition, they hold information like, e.g., due dates, available resources, documents and organizational or role assignments. Additionally, knowledge workers should be able to add constraints and conditions to it. Each process consists of linked tasks trees and/or tasks so that knowledge workers can coordinate their efforts.

Task trees consist of several tasks and possibly further sub-task trees, giving the workers a recommended order for performing their sub-ordinating tasks, though not being mandatory. The tasks within a task tree are supposed to be performed before the parenting task tree is completed, though this is not enforced. Task trees can be specialized as either to-dos list for prospective work, or as checklists for retrospective work and quality assurance.

Task Tasks in the proCollab system consist of a description of the work to be done, a current state and the assigned user or role. Further, optional conditions can be added in verbose form.

Template These do supply knowledge workers with means to comfortably pre-specify plans for future work and can help with planning and re-utilization of iteratively refined work-steps or phases. Templates can be instantiated into instances, which then can be worked with. There are process, task tree and task templates.

Instance Workers collaborate at collaboration run-time based on instantiated proCollab components/entities. There are process, task tree and task instances.

The listed task trees and processes are also referred to as contextual stateful entities (CSEs), as they span open a context for themselves and their sub-ordinated child entities or "sub-entities". In this work, the term *project* is used equally with the term *process*, as in task management systems the former is often represented through the latter.

Each of the key components of the proCollab system, namely the process templates, process instances, task tree templates, task tree instances, task templates and task instances are subject to proCollab's state management and can therefore be described as stateful entities (SEs). More details regarding the state model can be found in the succeeding Section 2.2.2.

2.2.2 ProCollab State Management

The state management employed by proCollab is an important tool in order to provide knowledge workers with the awareness they require. The information on state management presented in this section was acquired from [13].

Figure 2.6 illustrates the stateful entities introduced before and the reference state models, state models and state model instances used for state management. Note that the active states shown for the "State Model Instance" in the top-left give information on the stateful entity's current pre-defined state and it's current, user-defined sub-states.

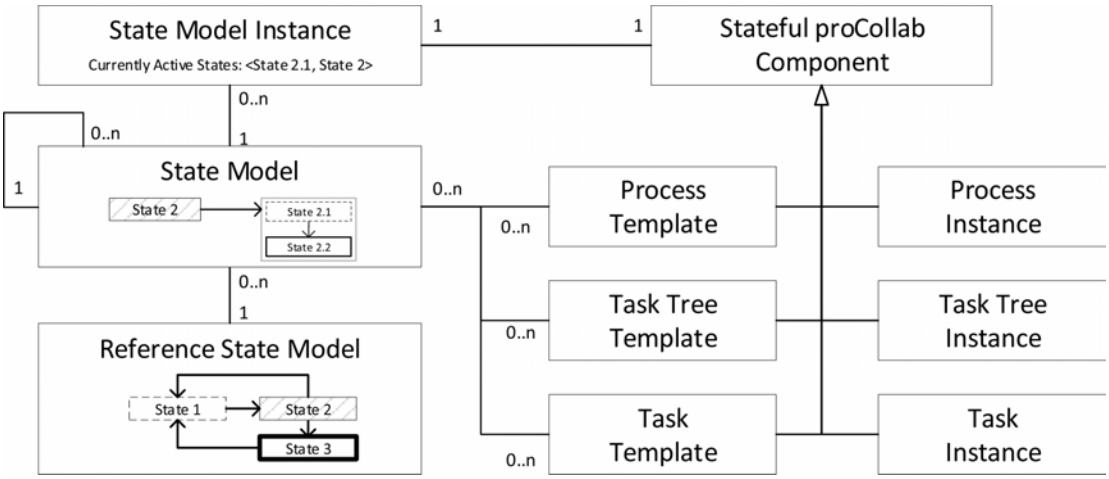


Figure 2.6: State management for entities in the proCollab system, reprinted from [13].

Reference state models do declare the states and the state transitions for a given type of entity. Figure 2.7 gives an overview of proCollab's reference state models for process instances, task tree instances and task instances. The states as shown in the figure can moreover be marked as being refineable into user-defined states.

State models are each linked to a specific reference state they are derived from and an optional parent state model. Furthermore, state models can be employed to refine a references state model's states. An example for refining the reference state model's "running" state can be observed in Figure 2.8: custom states originating from the V-Model as used, e.g., in automobile development are shown. If an entity is set into any of

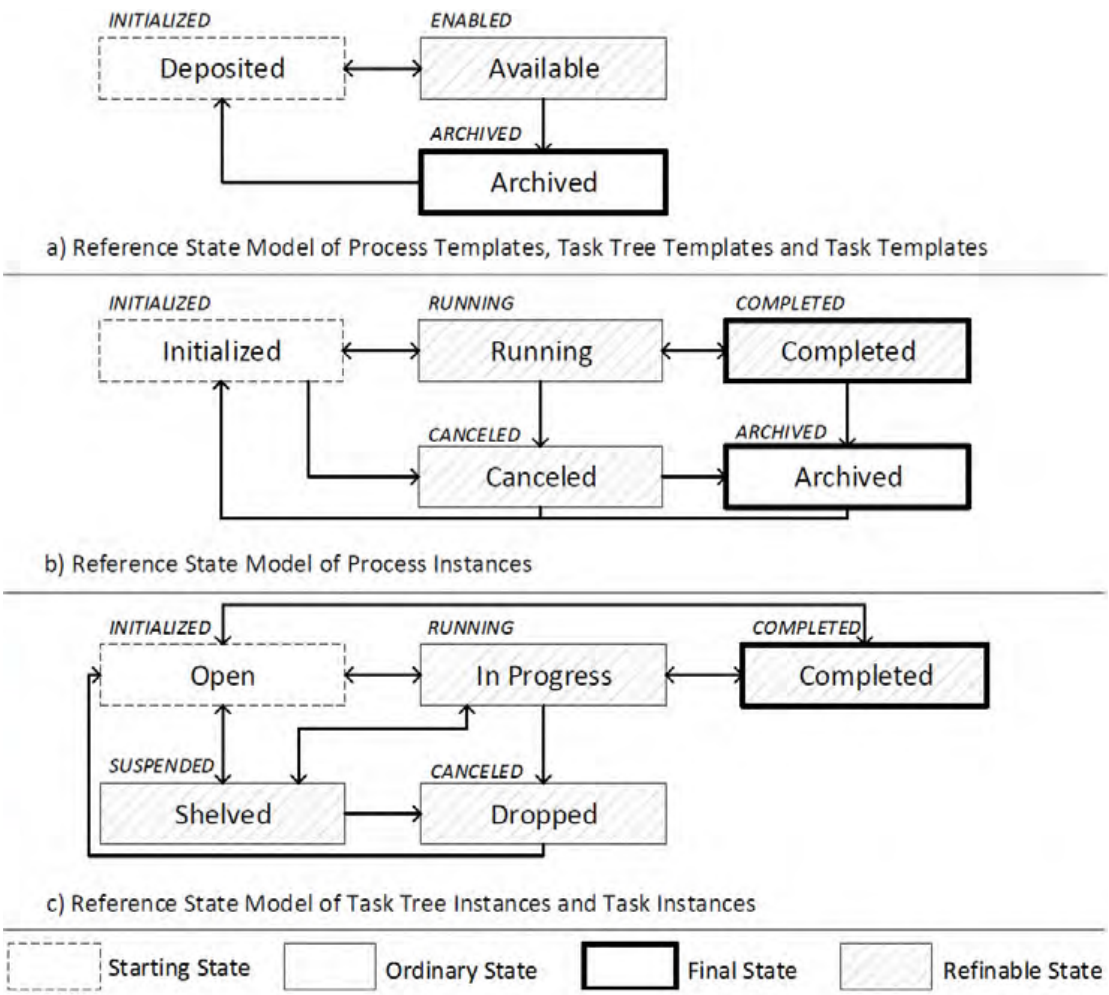


Figure 2.7: ProCollab's reference state models, reprinted from [13].

the substituting custom states, e.g., "Requirements Eng.", it will be considered being in the pre-defined state, e.g., "Running", as well.

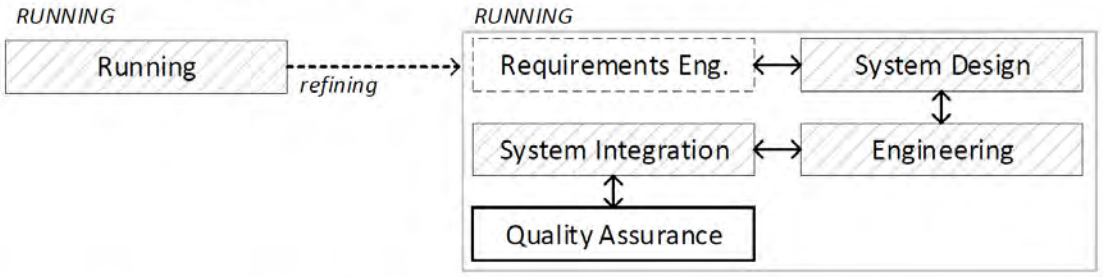


Figure 2.8: Example for a refined state in the proCollab system, reprinted from [13].

Finally, through state model instances the proCollab entities become stateful. As states can be refined, the state model instances list a sequence of currently active states, ranging from the most specific refined state to the most abstract state. The latter then being the state defined by a reference state model.

2.3 Use Cases for Constraints

Practical use cases for constraints arise with basically every scenario involving KiPs. Out of the manifold possibilities, a software development project was chosen to provide an example for the use of constraints in KiPs. The project focusing on the development of a website is taken and adopted from the proCollab system. This use case with many of it's tasks was also the scenario used for the Test-Driven Development of the implemented constraints (see Section 4.1). The scenario is laid out below, together with how its tasks would be formalized and organized in the proCollab system as seen through the client.

The agile approach oftentimes used to realize software projects can be seen as a good example for a KiP. Prominent features of agile software development are described by [14] as:

- Individuals and interactions over processes and tools.
- Working software over comprehensive documentation.

- Customer collaboration over contract negotiation.
- Responding to change over following a plan.

The characteristics for agile development are leading to uncertainty due to constant changes and adaptions as well as limited possibilities to plan ahead. Based on collaborative effort between the involved individuals, the software product gets iteratively refined and reworked on during the process. Showing these characteristic it becomes clear that a development process identifies as a KiPs.

Figure 2.9 shows an excerpt of this website development process, taken from the web client for the proCollab system. Visible on the left side is a to-do list to be used for prospective work and on the right side a checklist for retrospective quality assurance. The project, here being synonymous to the process, consists of stateful entities, e.g., tasks and task trees.

Constraining the task execution for such a development process can be used to express relationships between individual steps within the process. This allows for scheduling and coordinating efforts so that developers can successfully work towards their common and individual goals. Without going into the details of constraints, yet, some examples for dependencies of increasing complexity are given below.

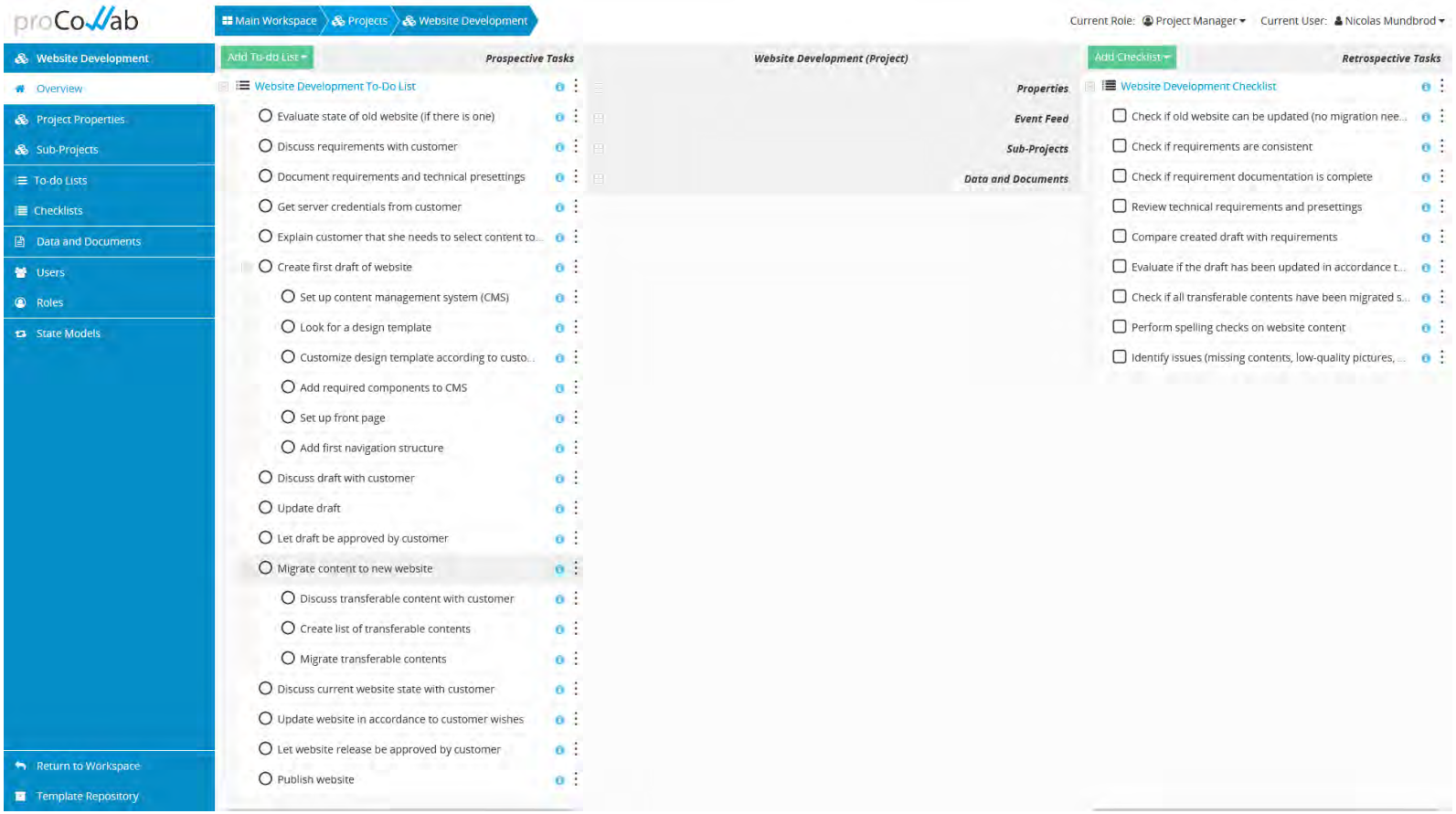


Figure 2.9: ProCollab project for the development of a website (example use case).

Example 1 As a first example, Figure 2.10 shows the dependency between two tasks of the to-do list: In simple terms, the new website can only be published (B) if the website release was previously approved by the customer (A). This is a very simple use case. Both A and B reside in the same context, being the to-do list as their immediate parent entity. Assuming that the dependency shown is the only one in the process, then, aside of A and B, no other entities are concerned by this dependency at any time.

Referring to states, the example could be more precisely phrased as follows: Work on B can only be started (or completed, proCollab states running or completed, respectively) if A was completed (proCollab state completed) at any time earlier.

Figure 2.10: A simple dependency in the website development use case.

Example 2 For the next example consider Figure 2.11: After the draft for the website was updated (A, in state "completed"), it must be approved by the customer (B, in state "completed"). This means that if A was *completed*, the work on the context can not be considered to be *completed* as long as the completion of B is still outstanding. Regarding the context, it is possible to define either the to-do list as to be the context hindered to *complete* or to go even higher in the hierarchy and affect the process itself including the to-do list. The latter case could then also require a restriction on setting the checklist (omitted in the figure) to state *completed*, as well. The knowledge workers need to decide on how far the effect is to reach.

Figure 2.11: A dependency with broader effect in the website development use case.

Example 3 The example in Figure 2.12 illustrates a case of even higher complexity: After the requirements and technical presettings are documented (A, in state "completed"), a check for the completeness of the requirements has to be performed (B, in state "completed") immediately after. Similar to example 2 above, this means that if A was completed, the work on the context can not be considered to be completed as long as the completion of B is still outstanding. The knowledge workers again need to decide on how far the effect is to reach in regards to the context. But in addition, the required, immediate follow up of B demands that no other entity is to be, e.g., started or completed in between. This automatically covers all sub-entities in the chosen context(s). These further entities get restricted implicitly, though, as new entities can be added to a context at any time. The effect such a dependency could have is therefore far greater than in the previous example. It could possibly effect every single work-item in the process. Another aspect pointed out in this example is that dependencies could exist between entities of different lists, e.g., an entity of the to-do list and an entity of the checklist. The effect could be decided to span over either only one of the lists or both of them. In the latter case this means that, in turn, the effects could affect knowledge workers of different contexts, e.g., different departments.

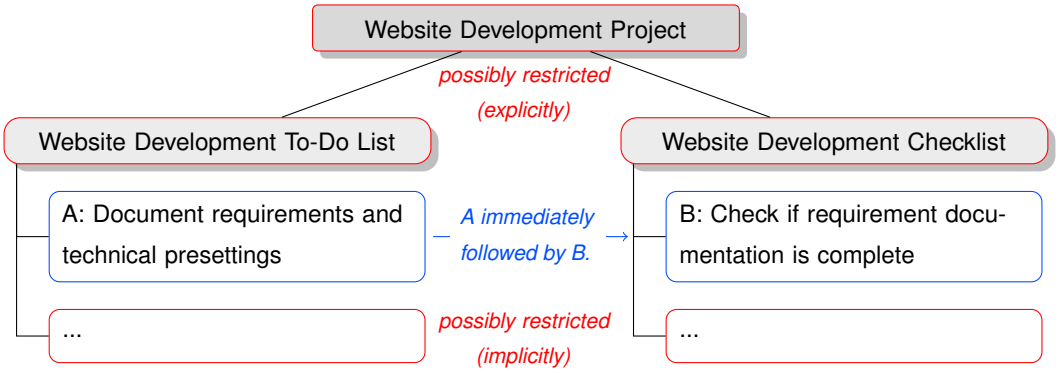


Figure 2.12: A Succession Constraint in the website development use case.

To sum up the given examples, dependencies in this work

- describe relationships between stateful entities from the same or from different lists/contexts,
- can spread their effect over a wider context and several work-items at once,
- cause their effects depending on the states of their entities, and
- affect entities by restricting state changes for them.

The complexity coming with dependencies is therefore high and would be increased even further when having an entity concerned by more than one dependency.

2.4 Related Work

Additional information was obtained by studying work and research related to the subject at hand. Section 2.4.1 provides an overview of a number of task management systems. After that, Section 2.4.2 discusses various work related to constraint-based approaches for expressing dependencies and relationships.

2.4.1 Task Management Systems

Various information systems exist that aim at helping with task management and team collaboration. A selection of task management systems that lends themselves to be used for supporting KiPs is introduced in this section. While many more features are supported by the systems than the ones presented in the following, the focus here is on the features being the most relevant to this work. The different task management systems are described by having a look at

- their work-items and means for hierarchical organization of these items,
- available states for the work-items, and
- supported task dependencies.

The terms *status* and *state* are of equivalent meaning in the following. Note that none of the task management systems presented in this section supports refinable states.

ActiveCollab

ActiveCollab focuses on task management, team collaboration, time tracking and invoices for customers [15]. It can be extended with additional functionality through add-ons.

Work-items and organization Work with ActiveCollab centers around *tasks* assigned to *projects* and optionally grouped in *task lists*. Furthermore, tasks can be refined into *sub-tasks*. Both tasks and sub-tasks can be filed under *milestones*. Figure 2.13 shows the ActiveCollab user interface with several tasks organized in task lists.

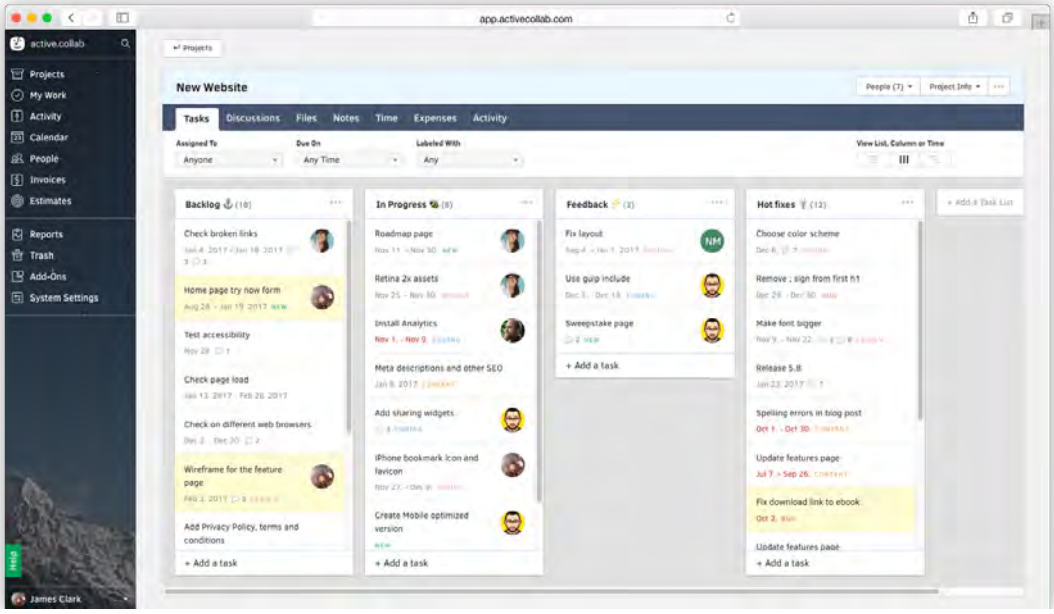


Figure 2.13: ActiveCollab user interface with tasks, reprinted from [15].

States Projects, milestones, tasks and sub-tasks can be *marked as completed*. In addition, users can create *custom labels* for projects, tasks and sub-tasks. Through *labels* it is possible to annotate the status of the work performed on the item, thus

allowing for a sort of user-created states. Task lists cannot be marked as completed and do not have labels available.

Task dependencies Options to create dependencies between tasks are currently not available but are *planned as an upcoming feature* [16] for communicating the execution order of tasks. One of the currently available "soft" ways to advise workers on where to focus their efforts is, e.g., highlighting tasks that are to be prioritized.

Asana

Asana [17] is aimed at helping to coordinate the work of a team by sharing tasks and project related information. It offers base functionality that can be extended with numerous plug-ins.

Work-items and organization *Tasks* are at the core of Asana and can either be assigned child entities of one or more parenting *projects* or unassigned (loosely existing) work-items. Furthermore, tasks can be refined through *sub-tasks*. Being similar to task lists, *sections* inside a project allow to group tasks together.

States Tasks can be set in one of two states: the default *state incomplete* and *state completed* (cf. Figure 2.14: note the tick marks left of each task). Custom fields of tasks can be used to attach further information, e.g., progress could be tracked by fields named "in progress" or "in review" (cf. Figure 2.14: listed in column "Status"). Sub-tasks can be set into states incomplete and complete as well, but do not have custom fields. For sections, there are neither states nor custom fields. Projects do have their *current state marked by color* but do not have custom fields available.

Task dependencies Asana supports a simple form of task dependencies: *tasks can be marked as waiting on other ones*. First, this form of task dependency can be used to express that a single task is waiting for another single one. In addition, it can also be used to mark multiple tasks to be waiting for another single one, or to mark a single task to be waiting for multiple other ones. A banner for the waiting task(s) will show information on the status of the preceding task(s). As soon as

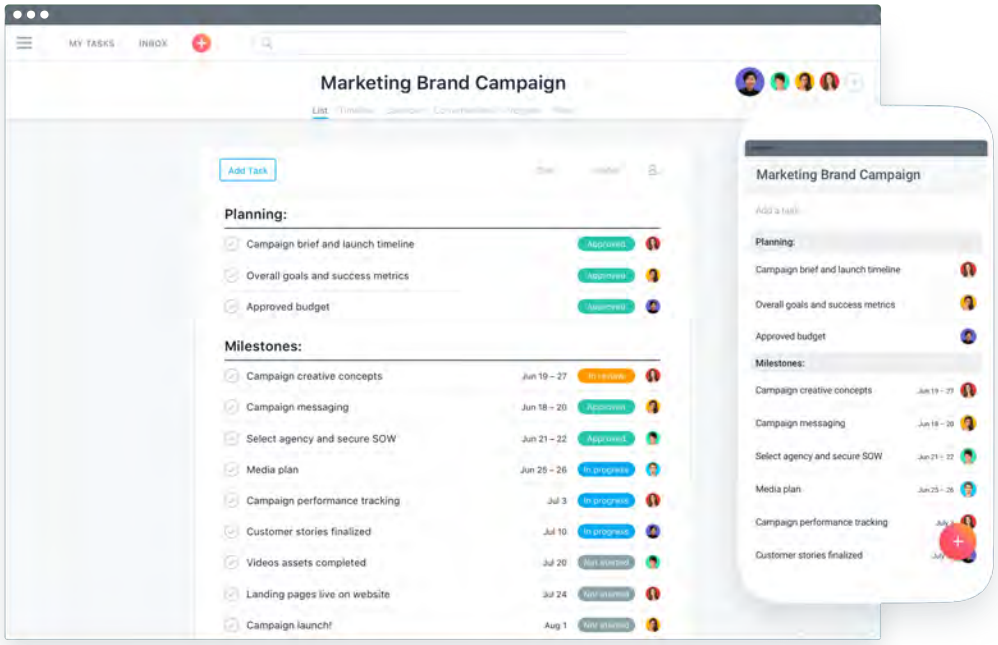


Figure 2.14: Asana user interfaces for desktop and smartphone, showing tasks with states and custom fields, reprinted from [17].

all preceding tasks are set to state completed or the preceding tasks have their due date (one of the default fields available to tasks) changed or removed, the assignee(s) of the waiting task will receive a notification.

Basecamp

Basecamp [18] is a system for supporting project management and team communication. It combines several tools for, e.g., task management, messaging and communication as well as to-do and event scheduling. The following information was obtained from [19].

Work-items and organization Tasks, named *to-dos* by Basecamp, are part of *projects*. *To-do lists* created within a *project* allow to further organize *to-dos*. In addition, *groups* created in to-do lists can be used similar to sub-lists for grouping to-dos together. To-dos can be assigned to milestones. Figure 2.15 shows a to-do list that has two groups created within ("Awaiting Fix" and "Fixed").

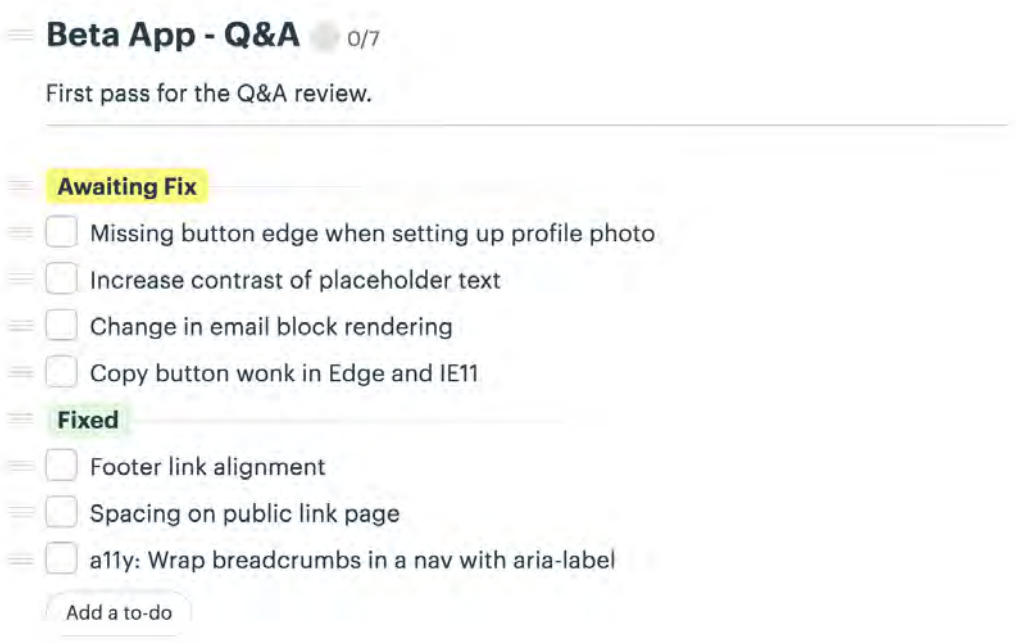


Figure 2.15: Basecamp to-do list with groups, reprinted from [20].

States The to-dos have two states for marking them as *completed or uncompleted*. Milestones can be marked as completed as well. To-do lists are stateless and organizational units only, though they display a *counter showing how many of the to-dos assigned to a list have been completed*. Groups are used as headers for visually highlighting to-dos that belong together, and are stateless. Projects can have their status set to *active*, *on hold* or *archived*. In addition, the milestones offered by Basecamp can be marked as *completed* as well. Means to define any form of user-created states are not available for any of the entities mentioned above.

Task dependencies An option in Basecamp that comes close to task dependencies is to declare *a milestone's completion being dependent on the completion of certain to-dos*. To-dos can be assigned to a milestone by users, but will then only be listed as being connected to the milestone. No restrictions on work will arise at any time based on this. Furthermore, no form of automated notification will be given when all to-dos assigned to a milestone have been completed.

Wrike

Wrike is described as "Cloud-based collaboration and project management software" [21].

Work-items and organization Wrike has *tasks* and *sub-tasks* assigned as children to *projects* and *sub-projects*. *Milestone tasks* are available as a specialized form of tasks. *Folders* and their *sub-folders* are organizational units that can be used for grouping projects, sub-projects, tasks and sub-tasks in them.

States Tasks, milestones and sub tasks can be set to one of *several statuses*. Processes and sub-processes have their *own set of statuses* available to them. Folders and sub-folders are stateless. Wrike offers *workflows* that can be used to define which status a work-item can or should be set into, depending on the current one. Users can customize workflows and create additional statuses. Figure 2.16 shows the default workflow with its 4 groups: active, completed, deferred and canceled.

Statuses can be added to each of these groups. A work-item can, e.g., only be changed into a status of group deferred if the current status of this work-item is from group active. While sub-states are not available in Wrike, statuses belonging to the same workflow group are in a similar child-parent relation to this group as sub-states to their corresponding parent state.

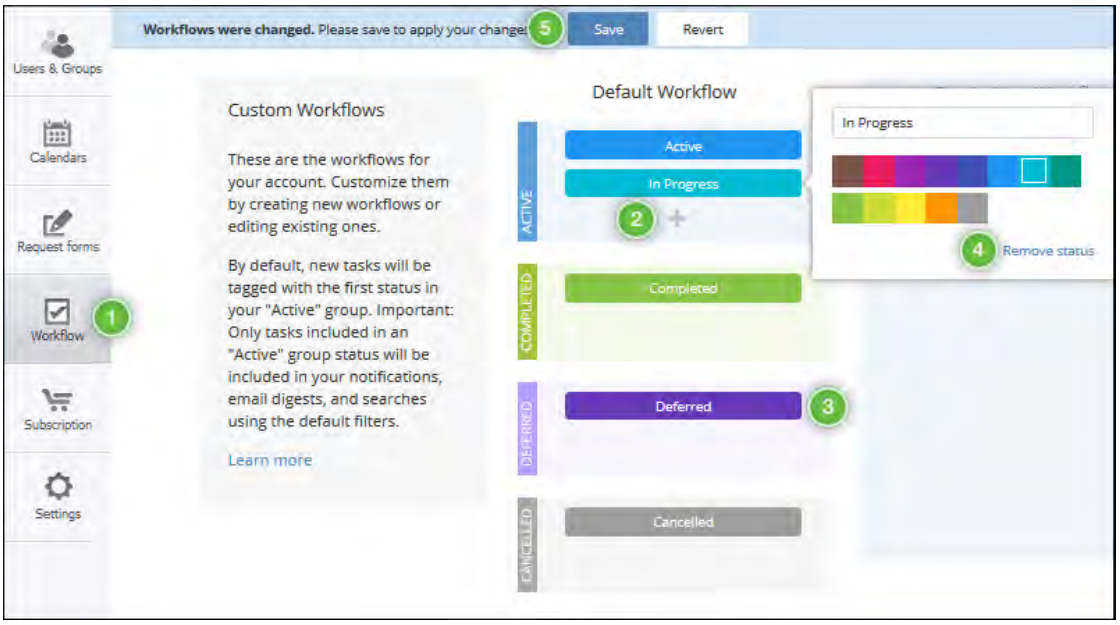


Figure 2.16: Wrike default workflow with customizable elements, reprinted from [22].

Task dependencies Four types of task dependencies are supported by Wrike. They enable users to define finish-to-start, start-to-start, finish-to-finish and start-to-finish relationships between tasks. These dependencies allow to set the order in which tasks should happen (e.g., `task 1`, then `task 2` and so on) and lead to two features of automation. First, whenever the start or end date of, e.g., `task 1` is changed, the start and/or end dates of the following tasks are automatically changed accordingly. And second, when, e.g. `task 1` is completed, the user to whom `task 2` is assigned will receive a message stating that work on `task 2` can now begin.

Comparison of the Task Management Systems

As constraints for dependencies can be created for SEs and for such only, the presented task management systems including the proCollab system (see Sections 2.2.1 and 2.2.2) are first compared by laying out their stateful entities and stateless containers. Table 2.2 provides the corresponding overview.

	Stateful Entities	**Stateless Containers**
ActiveCollab	sub-tasks, tasks, milestones, projects	task lists
Asana	sub-tasks, tasks, projects	sections
Basecamp	to-dos, milestones, projects	to-do lists, groups
Wrike	sub-tasks, tasks, milestones, sub-projects, projects	folders, sub-folders
proCollab System	sub-tasks, tasks, sub-task lists, task lists, sub-processes, processes	-

Table 2.2: Task management systems compared by their stateful and stateless entities.

Each of the task management systems offers several stateful entities for performing work. Many of these stateful entities can also span open a hierarchy and therefore can be used to define the context of constraints. As an example, if a sub-process is selected to be the context of a constraint, this constraint's effects would only affect this very sub-process and any tasks or sub-tasks assigned to it.

Dependencies are not (yet) supported by each of the systems and the ones found are mostly few and simple ones. Room for extending the number with additional constraints is plenty. In the case of ActiveCollab, the support of not yet-existent task dependencies can already be found on a roadmap for planned, future features.

2.4.2 Constraint-Based Approaches

Declare Workflow Management System

An important and major contributing factor to the present work is the Declare system developed by Pesic et al [23]–[26]. Declare is described as a constraint-based system for modeling and supporting loosely structured processes by employing a declarative language and temporal logic [23]. Being a workflow management system (WFMS) [24], the focus of Declare is on modeling and automatically executing processes while granting users the option to make changes to the execution of activities and (as referred to in later papers) tasks. Dependencies between tasks are expressed by applying constructs Declare refers to as "constraint templates" [23]. The three main components of Declare are

- a tool for modeling processes, called the Designer;
- the Framework used for process enactment and for changing models at runtime and
- a tool named Worklist that is employed to execute processes and see recommendations for their execution.

Figure 2.17 shows these components and symbolizes the users, able to influence and change the process execution.

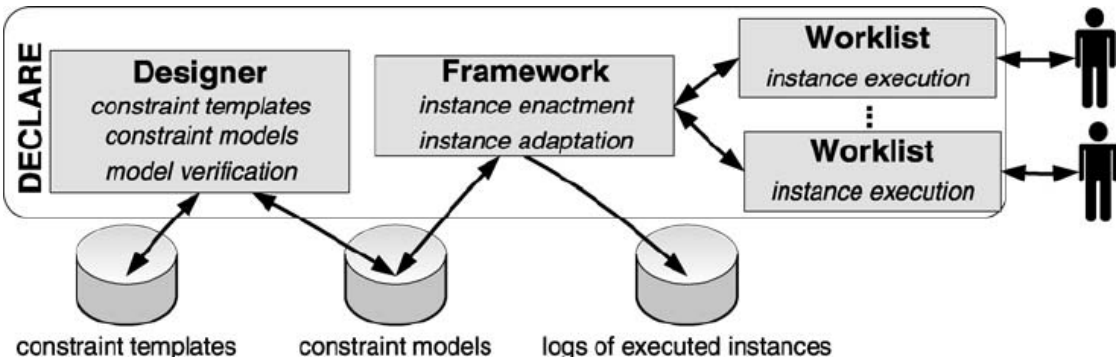
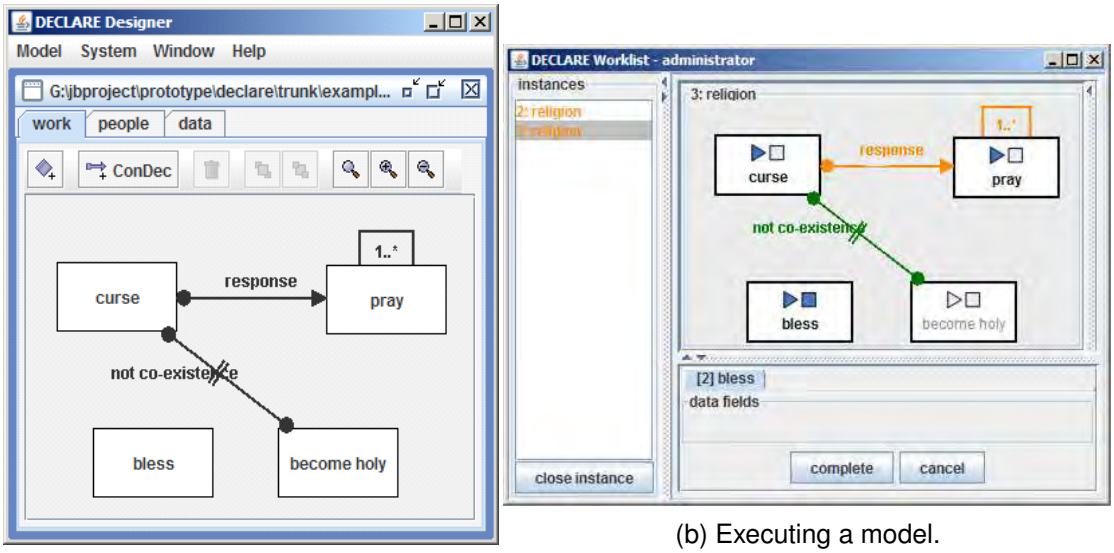


Figure 2.17: The architecture of Declare, reprinted from [25].

A couple of similarities can be seen between the proCollab system (see Section 2.2) and Declare. First, both aim to support loosely structured processes. Second, Declare provides a high amount of flexibility and allows for modifications to the activities and constraints during run-time [23]. Flexibility is a focal point of the proCollab system as well. In addition, both employ tasks which are operated on by the workers [24]. Another very important similarity between both systems is that each by default does not have any restrictions (aside of authorization and access control by proCollab) for the work performed on tasks. Only through the addition of constraints the system has rules added which users must or should follow when performing their work.

Similarities aside, the ways in which Declare and proCollab support KiPs are fundamentally different. With Declare a good amount of the work done by its users is spent during design-time for modeling of processes and setting up constraints. Figure 2.18 shows the user interface for creating a constraint-based model. On the left side a process model and its tasks is seen, with visualized constraints between them. Users get presented this model (seen on the right) and then execute the tasks, in accordance to the meaning of the constraints [24]. The proCollab system on the other hand is a task management system. With it, users spent most of their time not with models but on working with task lists and tasks. Hereby, for proCollab the driving factor for progressing through the process is not a model-based execution with a heavy reliance on pre-created constraints. Instead, the driving factor are user-induced changes to the various states of the work-items for accurately monitoring the progress of the process.



(a) Creating a model.

(b) Executing a model.

Figure 2.18: Creating and executing models with Declare, reprinted from [24].

Constraints are needed by both systems to provide guidance for the decision-making of their users. Reasoning and conceptual ideas for supporting constraints provided by Declare are therefore relevant and influenced this work. First, the work based on Declare lays out a number of constraints and gives a thorough explanation of their meaning [26]. The plan on adding temporal constraints is mentioned as well [24], [25] (see Problem 6, Section 1.1). And second, it highlights the need for guidance instead of strict ruling (see Problem 5, Section 1.1) and presents a flexible approach for achieving this (see Problem 2, Section 1.1) [23], [25].

Critical Path Method

Critical Path Method (CPM) is a method for scheduling a set of project activities [27]. It calculates the longest path of activities to the end of a project and in addition the earliest and latest date at which each of these activities can finish without delaying the project. Activities which cannot be delayed without also causing a delay of the project are called *critical activities*. The sequence of activities that takes the longest to complete and only consists of critical activities is called the *critical path*. CPM helps by identifying critical

paths so that activities can then be prioritized accordingly with the goal of reducing the project's duration.

The essential requirements to denote a critical path for a project are the activities, their duration and the dependencies between them. The latter are used to put the activities into sequence. Having constraints for expressing such a dependency and sequence available is therefore an important necessity to apply various methods for project management. The type of constraint applied here, requiring that an activity can only be started after the preceding one was completed, is called a *precedence constraint*.

Some of the merits of applying the CPM as taken from [27] are:

- identifying where parallel activity can be carried out,
- calculating the shortest amount of time in which the project can be completed,
- assessing the sequence of activities and their timings, and
- conceiving task priorities.

CPM provides an example for an important and very commonly found type of constraint [28] (see Problem 6, Section 1.1). These precedence constraints allow to define sequences of activities or tasks and help determining where parallel work may occur.

Precedence Diagramming Method

Precedence Diagramming Method (PDM) creates a visual representation of the activities in a process [29]. Similar to CPM its purpose is to help with scheduling projects through relationships between the activities and determining critical tasks. It depicts the activities as nodes and their relationships as arrows connecting the nodes. PDM adds a number of additional dependencies to the single one used in CPM. The types of dependencies used for PDM are bound to certain states of the activities, *start* and *finish*:

- Finish-to-Start (FS, used in CPM as well).
- Start-to-Start (SS).

- Finish-to-Finish (FF).
- Start-to-Finish (SF).

Another difference to CPM is the addition of lead-lag-factors for the dependencies. Through these it is possible to define the minimum amount of time between the start or finish of an activity and the start or finish of another preceding one.

While being able to express a broader and specifically state focused selection of dependencies is helpful for sequencing activities, such dependencies also increase the complexity when working on the process. The key point taken from [29] is to be wary of the effects when offering state focused dependencies for sequencing tasks. One problem is the creation of loops when multiple dependencies are declared for the same activities [29]. In some cases, expert insight and an overview of the grander scheme of interdependencies is advisable when setting up multiple and partly interwoven dependencies (cf. Problem 1, Section 1.1).

Temporal Constraints

The support of temporal constraints and time patterns in process-aware information systems is discussed in [30]. Based on empirical evidence, 10 time patterns are suggested and existing process-aware information systems are examined for their capabilities to support these. For each pattern, formal semantics are provided and existing approaches for the patterns' support are evaluated.

An example for one of the patterns is a time lag between two activities. In addition to the minimum time values similar to the lead-lag-factors in PDM (see Section 2.4.2), also proposes a maximum value and a time interval with a min-max-range is proposed by [30] as well. A second example for a time pattern given by [30] allows "[...] to restrict the execution of a particular element by a schedule; e.g., a timetable (e.g., a bus schedule)." This could mean restricting certain state changes during a set timespan.

The patterns offer an overview over various types of temporal constraints and lay out a concept for viable further extensions of this work (see Problem 6, Section 1.1).

2.5 Requirements

Looking at the insights gathered from analyzing related work and research done for proCollab, a number of requirements for supporting constraints and dependencies can be identified. The goal is to develop a concept for the support of clearly defined constraints and dependencies while adhering to the core functionality and traits of the proCollab system as an example for a task management system supporting KiPs. Overall, the concept has to be a generic one that can easily be applied as an extension to various task or process management systems.

Based on the challenges and requirements when supporting KiPs (see Section 2.1) and the system aspects of the proCollab system, a number of requirements listed in Table 2.3 can be derived.

Requirements for Supporting Constraints
Req 1: Constraints express dependencies of stateful work-items.
Req 2: Focus on states and state changes.
Req 3: Constraints are context aware.
Req 4: Constraints provide guidance.
Req 5: Provide flexibility and adaptiveness to frequent changes.
Req 6: Limit complexity to minimize the potential for errors.
Req 7: Provide means to set contextual, automated behavior.

Table 2.3: Summary of the requirements for supporting constraints in KiPs.

The descriptions below explain the reasoning behind the mentioned requirements and give their detailed meanings.

Req 1: Constraints express dependencies of stateful work-items Knowledge workers perform in a KiPs based on stateful, task-centric work-items, e.g., tasks or processes. In order to aid the knowledge workers with their decision-making, it is necessary to define the dependencies that exist for the work-items. Constraints are to express the dependencies for these work-items.

Req 2: Focus on states and state changes Progress on KiPs is monitored by updating the various states of the work-items as knowledge workers perform their jobs. In many task-management systems users have multiple states available which they can select from. Constraints are to monitor and react to changes of every state available to the work-items including user-created ones. The effects of constraints is to be based on the current state of the work-items. Users must be able to create constraints for any stateful work-item in the information system and select the states which a constraint is to react to.

Req 3: Constraints are context aware As multiple knowledge workers perform in different contexts, sometimes even being assigned to different departments of a company, constraints have to be wary of the contexts they are to be applied to. E.g., the effect of constraint that is meant to create a dependency for two tasks does often affect their parent entities and even the sub-entities of the latter (see Example 3, Section 2.3) as well. Stateful CSEs like, e.g., task lists in the case of proCollab, can form a context for constraints. There must be means to define the context(s) in which a constraint should work in so that the effect would not affect tasks it is not wanted to reach. Users must be able to define one or more CSEs that are used as the context(s) for a constraint during the constraint's creation.

Req 4: Constraints provide guidance Because of the frequent changes that KiPs undergo, knowledge workers have to stay adaptive as well and alter formerly made decisions. This could mean that knowledge workers, e.g., might need to disrespect a constraint that has already been created and perform a state change it actually means to prohibit. Instead of imposing fixed restrictions on state changes, constraints are to offer guidance to the knowledge workers. This is done by the constraint proposing which state changes are to be omitted at the time being according to the dependency expressed by the constraint. The task management system is to decide as wanted how strictly these proposals must be followed.

Req 5: Provide flexibility and adaptiveness to frequent changes As changes to KiPs and with it to work-items and constraints are normal and expected to come, constraints have to be flexible and adaptive. Constraints must be able to automatically

react to the creation, deletion and changes of work-items. Furthermore, users must be able to make changes to constraints, e.g., selecting a different CSE as a context for the constraint (see Req 3 above) with the constraint then automatically adjusting to the alteration.

Req 6: Limit complexity and unwanted interplays to minimize the potential for errors With a higher number of Constraint types and more fine grained functionality, the complexity of the system and number of possible problems will rise. This often means that users need to have a good understanding of the process' structure and the meanings of existing constraints and interdependencies when putting new constraints in place. A balance between the expressiveness of constraints and the usability of these constraints has to be found. This concerns the selection of available constraints as well as the functionality offered by them. Options to deactivating selected functionality have to be offered.

Req 7: Provide means to set contextual, automated behavior Small parts of the knowledge workers' work-flow is to be automated in order to help with making the system reflect changes. The goal of automating system behavior is to guarantee that certain actions and minor steps of the workflow are performed. Users should be assisted by automating work and actions, e.g., setting a task list to state "completed" as soon as all its sub-entities are in state "completed" as well.

2.6 Summary

In this chapter, groundwork and basics for the concept for constraints in KiPs were presented:

- The characteristics of KiPs were shortly presented as well as the general challenges and requirements when supporting KiPs.
- A brief overview of the proCollab system including its state management was given next.

- A website development project was presented as a scenario describing various use cases for constraints.
- Next, related work was discussed granting an overview and short comparison of different task management systems as well as a look at several constraint-based approaches.
- Finally, the requirements for supporting constraints in KiPs, derived from the gathered insights, were presented.

3

Concept

The concept for the support of constraints in the scope of a task management system is based on the fundamentals discussed in Chapter 2. While the proCollab system serves as the task management system at hand for the support of KiPs, the generic concept can be adapted to other task management systems built to support KiPs as well.

The contributions of this chapter can be seen in Figure 3.1. First, the basics of rule engines as the driving technology behind this concept are introduced in Section 3.1. Next, the concept of constraints and core of this work is laid out in detail in Section 3.2, followed by a catalog of constraints which are based on the concept in Section 3.3. After that, the challenges of achieving the adaptiveness that is required when supporting KiPs are discussed in Section 3.4, together with ways for achieving correctness and the desired system behavior. Finally, several examples of how to extend the concept for added features, functionality and constraint types are shown in Section 3.5.

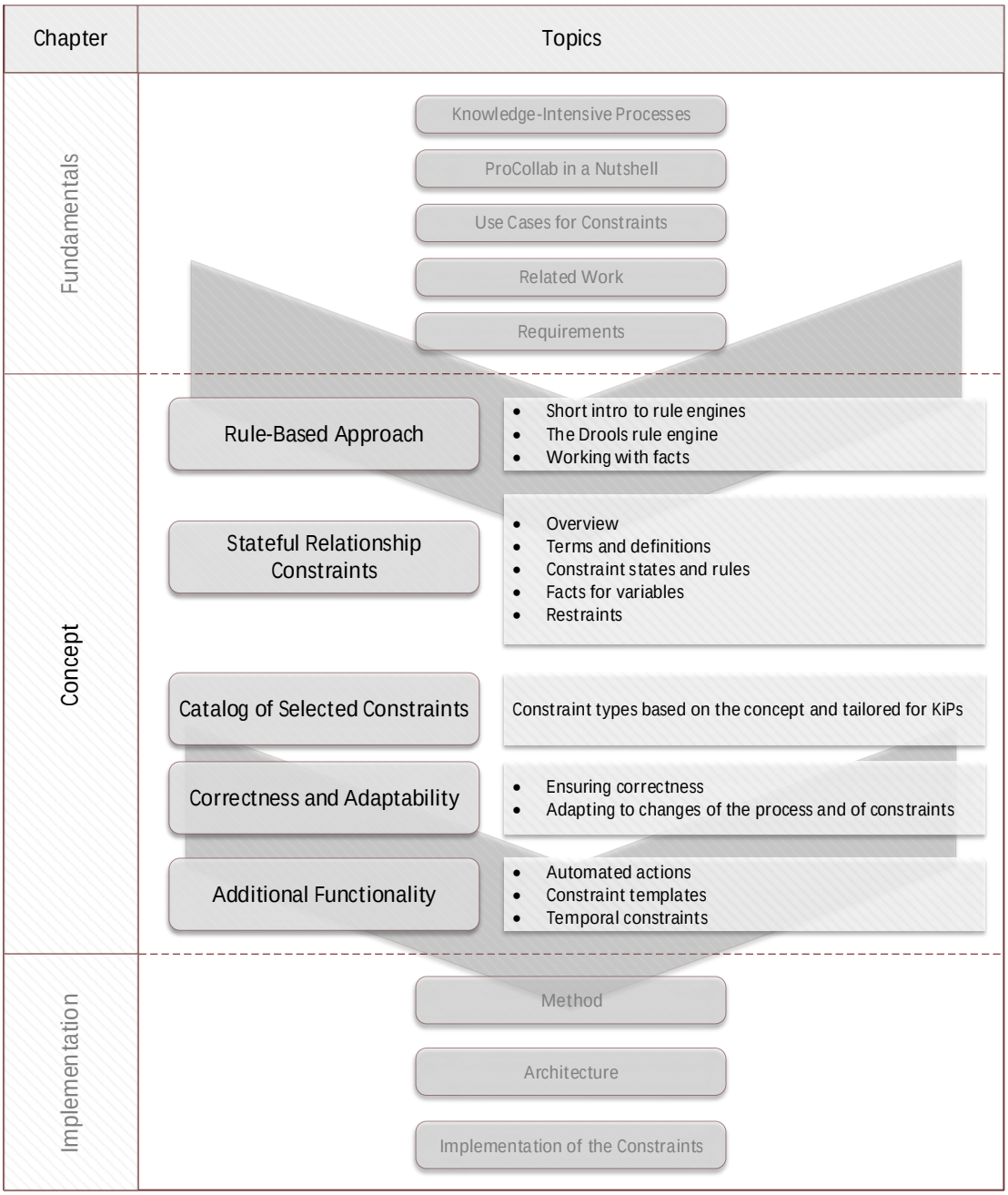


Figure 3.1: Roadmap highlighting the chapter on the concept.

3.1 Rule-Based Approach

Constraints are heavy on logic and perform their functions, e.g., proposing the restriction of state changes based on various conditions. A rule-based, declarative approach utilizing a rule engine to encapsulate the logic was chosen. The reason for the decision towards this technology as well its advantages and characteristics are laid out below.

In the broadest sense, a rule engine, also referred to as an inference engine, can be defined as:

Definition 6 (Rule engine)**:** *"Methods and apparatus, including computer program products, for inference processing in a fact-based business automation system, including receiving a rule set as a single package, generating a dependency graph for the rule set, and generating a sequence of processing logic for optimal processing of inputted facts" [31].*

3.1.1 A Quick Intro to Rule Engines

Rule engines encapsulate logic in rules separate from the regular code. Their rules are split into a left hand side (LHS) containing the rule's condition and a right hand side (RHS) defining it's effect. An example rule for the Drools rule engine is shown in Section 3.1.2.

Definition 7 (LHS)**:** *The left hand side of a rule containing it's condition. Specified by Drools' keyword "when".*

Definition 8 (RHS)**:** *The right hand side of a rule containing it's effect. Specified by Drools' keyword "then".*

Evaluation of a rule is done by performing comparisons of *facts* and their fields - a procedure called pattern matching. The facts are representations of objects made accessible to the rule engine by inserting them into the rule engines memory. All facts together form the fact base.

Definition 9 (Fact)**:** *Representation of an object made accessible to the rule engine so that it can evaluate on. All facts combined form the fact base. The Drools rule engine stores it's facts in the working memory.*

When a rule is fired due to the effect of another rule, this case is referred to as *rule-chaining*. Typically, this occurs once a rule inserts a fact on its RHS and another rule checks for it on its LHS. A second example could be one rule altering the value of a field that is tested by another rule.

Definition 10 (Rule Chaining)**:** *A rule's effect leading to the successful evaluation of another rule's condition. Thus the latter is then scheduled for firing. Rule chaining is a common mechanism of rule engines. Through it, a chain of rules ready to fire in sequence is established.*

Aside of chained rule executions, the order in which rules are evaluated is usually determined by the rule engine internally and usually not known in advance. Some rule engines like, e.g., Drools allow to set priority values for rules for when a certain order of evaluation is necessary. In general, the order of evaluating and executing rules is left to the rule engine.

Instead of having heavily nested conditional clauses of imperative code, the atomic rules of a rule engine stay easier to write and comprehend and can be easier to maintain in the long run.

3.1.2 Drools Rule Engine in a Nutshell

Due to it's maturity and broad number of accompanying tools and frameworks, the Drools rule engine [32] (Drools Expert) was the chosen as the key technology for supporting constraints.

An overview of the general Drools architecture is seen in Figure 3.2. Facts are inserted into the working memory from the Java side so that Drools has access to their information. The rule evaluation is done automatically and implicitly by Drools' rule engine. If it notices that a fact was changed or added that is relevant to a rule, the rule gets evaluated again

for the fact in question. This is advantageous when the need arises to react to changes of the proCollab system's entities (see Section 3.4). A positive outcome of a rule's evaluation then leads to this rule being scheduled for execution and added to the agenda. Only on an explicit call to Drools' firing function the rules currently waiting for execution in the agenda will go into effect and have their RHS executed. Drools utilizes an improved *Phreak* algorithm [33] that offers optimizations, e.g., sharing patterns between rules so that accessing shared facts and their fields only has to be done once. For more insight into the Drools rule engine and Phreak's mode of operation please refer to [32].

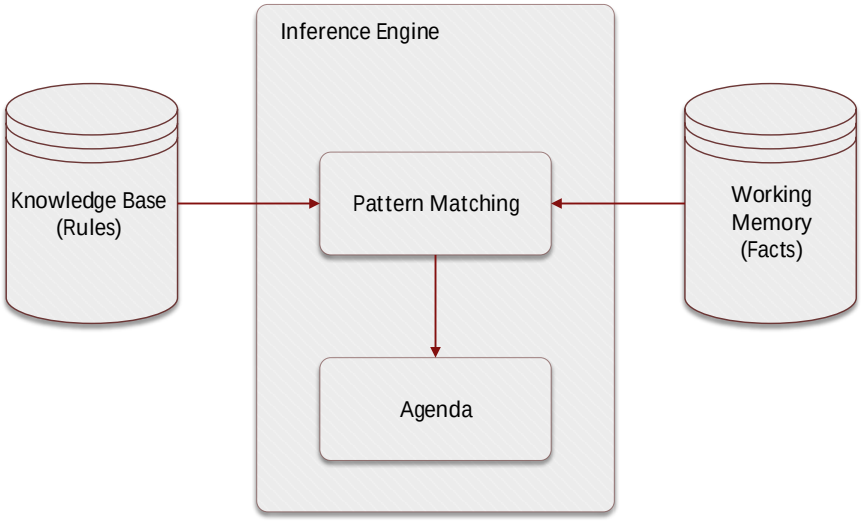


Figure 3.2: Simplified view of the architecture of the Drools rule engine.

Most of the tools related to the broad Drools framework, e.g., a web UI for rule creation and management, are geared towards enterprise business systems and were not used for the present work. Despite this, it can be seen as helpful to select a sophisticated base with ample opportunities for extension. More importantly, Drools has a years long and still ongoing history of professional development behind it and comes with a solid, proven foundation as well as broad functionality surrounding it [32]. For simplified coding, it allows to write rules in it's own *Mvel* language, in addition to the Java code that can be used when and where needed. An example rule taken from the official Drools documentation is shown in Listing 3.1 [32]. It tries to match against any facts of type "Applicant" and type "Certificate" in the working memory. The rule fires once

for each combination of an applicant with age less than 18 and a certificate referencing the applicant's id. The effect states an alteration of the matched fact. Fields used in the condition are referred to by Drools as "constraints" of the fact - a widely used term.

```
1  rule "Is of valid age"
2      when
3          $a : Applicant( age < 18, $id : uid )
4          Certificate( ownerId == $id )
5      then
6          $a.setValid( true );
7  end
```

Code Snippet 3.1: Example rule taken from the official Drools documentation [32]

A session, to which the rules and the working memory with the facts are assigned to, is created for the Drools rule engine. The type of session used for the concept is referred to by Drools as a *stateful knowledge session*. After being created, these sessions remain active and are continuously ready to evaluate the rules' conditions as needed. Evaluation happens once after the creation of a rule and each time a change is done on a fact that is used in the rule's condition.

The core aspects of the Drools rule engine can be described as follows:

- Like all rule engines it follows an declarative instead of imperative approach,
- encapsulates logic in rules,
- splits the rules into a condition (the LHS) and an effect (the RHS),
- evaluates the conditions based on facts stored in the working memory of the rule engine,
- automates the (implicit) rule evaluations,
- performs the firing of rules on explicit commands, and
- optimizes rule evaluation even when handling a very high number of rules and facts.

3.1.3 Working with Facts

Establishing and maintaining an up-to-date fact base, the sum of all facts, is an important requirement when working with rules. Facts are automatically persisted by Drools together with the session.

For working with constraints, the system's entities for work-items are on Drools' side represented trough facts. These SEs are usually quite heavy and with a greater amount of dependencies to other objects and collections. To incorporate information about the SEs into the rule engine a more compact fact type called *entity fact* was created. Entity facts only contain information necessary for the rules and constraints. Despite that Drools only inserts references to the objects instead of creating complete copies, more complex objects, especially nested ones, can lead to a decrease of the rule engine's performance. For this, the mentioned extraction of relevant data into more tailored facts is advisable. Additional created fact types like, e.g., for the constraints' variables, are described in Section 3.2.4.

Definition 11 (Entity Fact)**:** *A fact type representing an instance of a proCollab system object in it's current state.*

Entity Facts need to be kept up to date in order to constantly reflect the actual entities in the system. Ensuring that they remain valid can be done by having entity listeners or interceptors watch over objects and methods. In case of newly created entities, new facts get inserted. Updates cause a look-up for the corresponding, already existing fact and bring it up-to-date. And finally deletions of entities lead to removing the corresponding facts from the working memory. Rules are re-evaluated when changes are done to facts that are used in the conditions of these rules (see Section 3.1.2). This automatic re-evaluation allows for hands-off adaptability of the constraints. More on reacting to changes can be found in Section 3.4.2.

Collections, when directly inserted as single facts, would not allow the rule engine to handle them with the best possible performance and optimizations. Instead, the rule engine couldn't see changes done to individual elements of that collection and only notice these as changes to the collection as a whole. Then, all rules trying to match

any element of the collection in their conditions would be re-evaluated. With single facts for individual elements, the changes could be traced down to only this one fact and re-evaluations kept to a minimum. As an example, if a work-item had the ids of a number of assigned knowledge workers stored as a collection, it would be best practice to insert a single fact for each worker id of this collection instead of a collection as a whole. The facts holding the worker ids would then also reference the work-item, being the parent object of the original collection. Rule engines are somewhat similar to databases in this case, which do employ foreign keys for related objects.

3.1.4 Summary

A brief overview of rule engines in general and the chosen Drools rule engine were given in this section. This was followed by a discussion on working with facts and maintaining the fact base.

The key points of this section can be described as below:

- Rule engines allow to encapsulate logic in rules.
- They work on comparing facts and their fields, stored in the rule engines working memory.
- Facts are part of a stateful knowledge session that stays active as long as the system is running.
- Rules get evaluated autonomously and on a successful outcome get scheduled for firing (execution).
- On changes to facts or fields of facts that are used in a rule's condition this rule will be evaluated again.
- One rule's effect leading to the positive evaluation of another rule is called rule chaining.
- Firing rules and therefore executing their effects is done on explicit method calls.
- Facts for representing system entities are managed and kept up-to-date by an entity listener and interceptor.

3.2 Stateful Relationship Constraints

Constraints in this work are representing relationships between task-centric work items. Likewise to the SEs they work on, constraints have a set of states available to them as well. They can therefore referred to as stateful relationship constraints. The following section formulates the core of this work and presents the concept for the stateful relationship constraints.

At first, Section 3.2.1 provides a short overview of the concept and it's three main aspects. Next, terms and definitions will be the topic of Section 3.2.2. After that, the three main aspects of constraints will be discussed: the rules and constraint states together with their interactions in Section 3.2.3, facts and variables for constraints and their use in the rules' conditions in Section 3.2.4 and restraints in Section 3.2.5. Finally, a step-by-step description of how constraints are worked with is given in Section 3.2.6.

3.2.1 Overview of the Concept for Constraints

The concept for supporting constraints is founded on three core aspects, presented in Figure 3.3. The constraint states and rules are at the center (see Section 3.2.3). Input to each constraint is given in the form of facts. These facts represent the variables for constraints and the entities of the task management system (e.g., stateful work-items and information on their place within the hierarchy of processes, task lists and tasks). The output of constraints are the restraints, set up and revoked by the constraints rules.

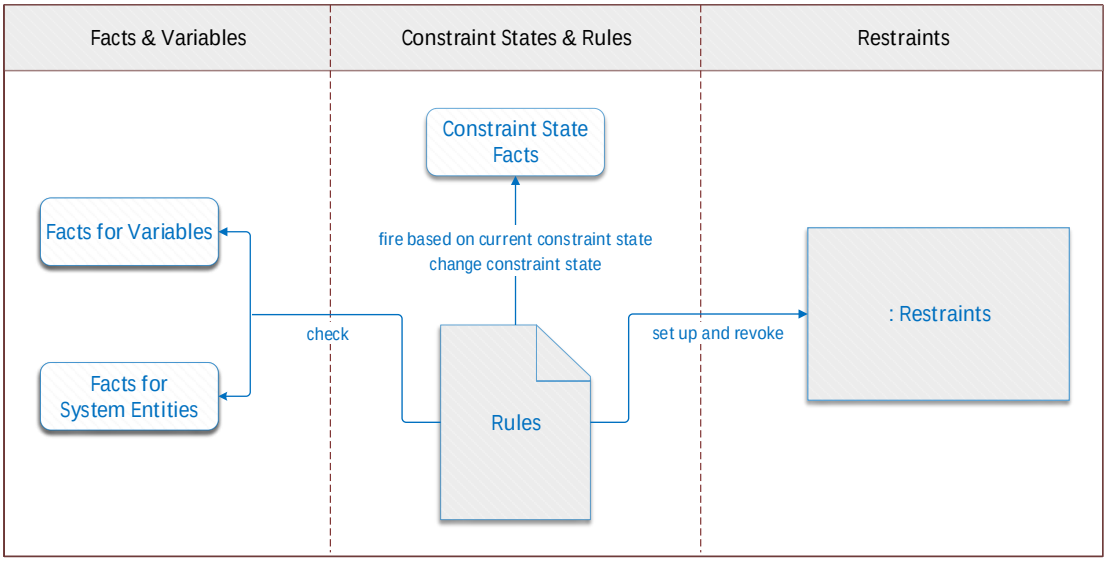


Figure 3.3: The three core aspects for supporting constraints.

The different types of constraints can be described through a common representation. Figure 3.4 visualizes this general concept.

Each constraint has a number of variables, set by the users during constraint creation. These variables specify on which SEs the constraint is to work on and which of the SEs states are relevant to the constraint. Additional entities affected by the constraints' effects can be automatically selected by the constraints, e.g., based on being sub-entities of a specified context (a task list or process; see Example 3, Section 2.3). Due to this, complexity does not only arise in the conditions shown for the constraints, but also can be found in the effects and in how far those reach. As the spread of the constraints' effects is not always easy to grasp, this concept therefore has users explicitly specify the context(s) for certain constraints. Note how the arrow of the constraint's visualization connects the state models of the two shown SEs. This highlights how constraints focus on the entities' various states.

Constraints have three stages through which they progress. Each stage is represented by a constraint state, *initialized*, *triggered* and *satisfied*. The logic for constraints is realized through several rules, each one taking over a smaller part of a constraints

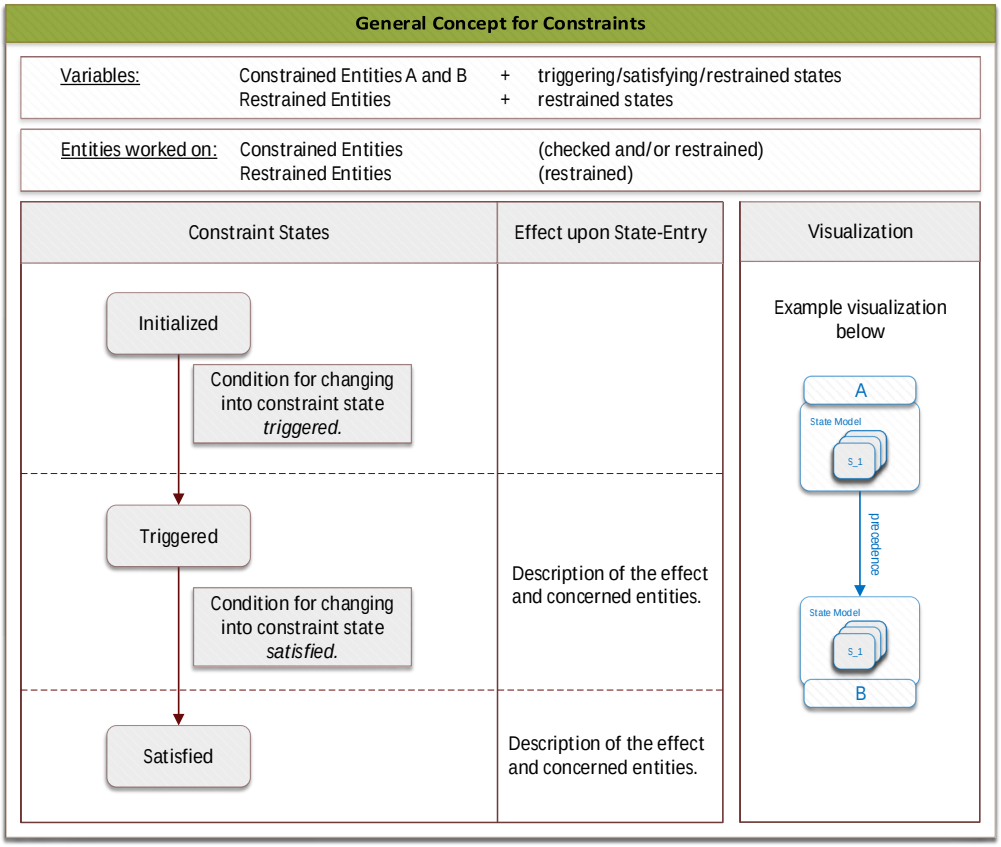


Figure 3.4: General visualization of the concept for a constraint.

functionality. Rules check for the current constraint state to determine if they are allowed to fire. Changing states is based on conditions, annotated on the connecting arrows. These conditions will be evaluated when the constraint is in the state denoted above the condition. If the condition is evaluated to be true, this then leads to the change of the constraint's state into the succeeding one. Effects like setting up or revoking restraints are caused automatically when a constraint is changed into the matching state. Through encapsulating the logic for constraints within compact rules, maintaining the code and functionality becomes easier. In addition, rules can be reused for different constraints and features that require equal or similar logic and effects.

Both the state changes and the noted effects are realized by rules (see Section 4.3.2). Instead of having new rules created for each concrete constraint, constraints of the same

type share an immutable rule set. The variable entities and their states for each concrete constraint are supplied by creating and inserting corresponding fact types (see Section 3.1.3). These facts are the distinguishing elements between concrete constraints of the same type.

3.2.2 Terms and Definitions

Variables are being set by users at constraint creation. Each constraint works on one or more SEs A. In addition, most constraint types also have a one or more SEs B set for the constraint. The constraint serves the purpose of representing a relationship for these A and B entities, like, e.g., a sequence in which these have to reach certain states. Entities A and B are referred to as Constrained Entities (CEs).

Definition 12 (Constrained Entity)**:** *An entity with a dependency expressed through a constraint. Selected by a user as either the A or B entity for a constraint and being monitored and/or restrained by it. Has triggering, satisfying and/or restrained states specified, selected from its state model..*

Furthermore, some constraints can have one or more SEs selected for them that only have restraints set up for. Such entities are called Restrained Entities (REs). Note that a CE with only restrained states specified could also be named RE. The terms do therefore also help with distinction.

Definition 13 (Restrained Entity)**:** *An entity either chosen by a user or automatically picked by a rule for being restrained by a constraint. Has restrained states specified, selected from it's state model*

CEs have one or more of their states specified as *triggering*, *satisfying* or *restrained* states. Depending on the constraint, states for one or more of these types have to be specified for each CE. REs picked by users need to have their restrained states set. The restrained states for automatically chosen REs are set by users as well, but uniformly for all of these entities. Whenever any of these types of states is to be chosen, users have the option to pick either a single or multiple states.

Definition 14 (Triggering state)**:** *A state specified for a CE that leads to the constraint being triggered (moving into constraint state "triggered") upon this entity being changed into it*

Definition 15 (Satisfying state)**:** *A state specified for a CE that leads to the constraint being satisfied (moving into constraint state "satisfied") upon this entity being changed into it*

Definition 16 (Restrained state)**:** *A state specified for a CE or a RE that specifies a state that this entity should not be changed into*

Due to their ability to cause a change of the constraint state, triggering states and satisfying states are also referred to as *goal states*.

Definition 17 (Goal state)**:** *A summarizing term used for triggering states and satisfying states as well*

The constraint states represent the current stage of a constraint and signal which rules are allowed to fire. A constraint in state *initialized* has not set up any restraints, yet. It will change into state *triggered* when a CE reaches a goal state. When *triggered* or *satisfied*, constraints both can set up or revoke restraints, based on the type of constraint at hand. A *triggered* constrained is yet to progress further into state *satisfied*.

Definition 18 (Constraint state)**:** *Used for signaling which rules are allowed to fire. When triggered or satisfied, depending on it's type, a constraint can set up and/or revoke restraints. The standard progression through the available constraint states goes from initialized into triggered and then into satisfied*

3.2.3 Constraint States and Rules

Constraint states and rules are tightly bound together, with the latter being part of the rules' conditions. This way, the constraint states are used to signal which rules can fire. Certain rules like, e.g., the ones for setting up and revoking constraints therefore can be written with only a minimum of lines on their LHS. The conditions of rules are checked to decide if a constraint's state is to be changed. Determining when to do so is of high importance to the correct functioning of the constraints and the main topic of this section.

The Meaning of the Constraint States

The standard progression through a constraint's states goes from *initialized* into *triggered*, and then into *satisfied*. The constraint states' meanings and caused effects are listed below.

initialized When *initialized* a constraints has not set up any restraints, yet. When the CSEs of the constraint reach a triggering state, the constraint will then move into constraint state *triggered*.

triggered A *triggered* constraint has become already active and caused an effect by at least setting up restraints, possibly also revoking restraints. For an example of a constraint that revokes restraints during the *triggered* state, please see the Simple Succession Constraint (SSuccCstr) in Section 3.3.3. While the SSuccCstr is *triggered*, it progresses through several sub-steps of this constraint state, first setting up restraints, then revoking these.

satisfied State *satisfied* can be seen as the final stage of a constraint moving towards fulfilling it's purpose. While for most constraints this means revoking their restraints here, for others it means setting up a last number of restraints. Restraints set up by constraints as they reach this state will remain in place, except reverting the constraint state to an earlier one is supported. An example showing a constraint setting up restraints when *satisfied* is given by the Completion Constraint (ComplCstr) in Section 3.2.3.

Different Cases for Moving Through Constraint States

In addition to the standard path of progressing through their states, allowing constraints to revert to former constraint states could be supported as well. This section will discuss cases for constraint state reverts and highlight the problems which can arise.

For differentiating between all thinkable state changes for constraints, Figure 3.5 depicts all different cases. Standard constraint state changes that are necessary for basic

constraint behavior are symbolized by solid lines. Non-standard cases, where the constraint state is reverted to a former one, are shown by the dashed lines.

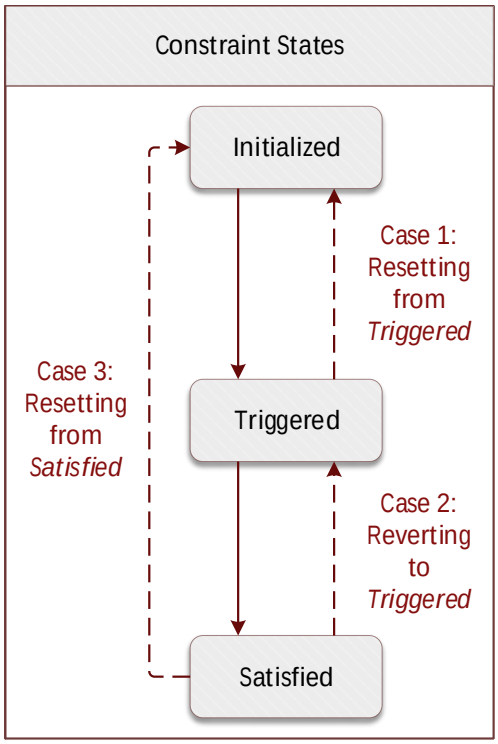


Figure 3.5: Standard and non-standard cases of constraint state changes.

Several cases of constraint state changes can be identified:

Standard Path Necessary for the functionality of the constraints and therefore always supported is the standard path for progressing through constraint states. This goes from constraint state *initialized* into *triggered* into *satisfied*. Upon reaching *satisfied*, a constraint will remain there.

Manual Constraint State Changes Due to usability and the frequent need for changes, which also concerns the constraints behavior, allowing users to manually change the constraints' states at any time is always supported, as well. Manually changing the current state of a constraint is considered to be a feature for special situations. Users, who have good understanding of the changes effects and decide to interfere

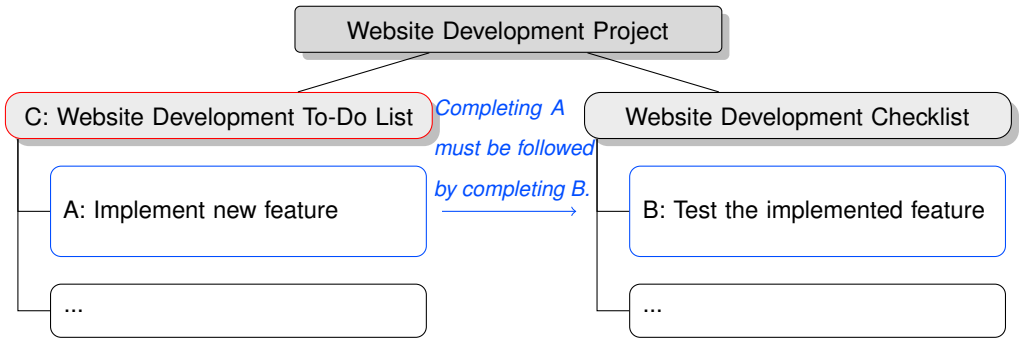
with the constraints on purpose can employ this feature to circumvent or adapt to problems.

Constraint state reverts The non-standard cases of reverting the constraint state include

- resetting the constraint state from *triggered* to *initialized* (case 1),
- reverting the constraint state from *satisfied* to *triggered* (case 2), and
- resetting the constraint state from *satisfied* to *initialized* (case 3).

The following part of this section discusses the cases for reverts, the possible problems and proposed solutions.

The Example Scenario for Constraint State Reverts To aid the following explanations, consider the example of a dependency as seen in Figure 3.6. An implementation task for a feature is to be followed by testing the implemented feature. For this, the goal states are set so that *completing* the implementation of the new future (A) requires the following *completion* of testing the feature (B). As long as B is not *completed*, the to-do list (CSE C) should not be allowed to be set into entity state *completed* as well. Note that this is a simplified example. Some scenarios could be way less forgiving when undesired reverts of constraint states happen.



Constrained Entity A, triggering state: *completed*
Constrained Entity B, satisfying state: *completed*
Restrained Entity C, restrained state: *completed*

Figure 3.6: Example scenario for constraint state reverts.

Case 1: Resetting from Triggered In the initial situation for this case, the constraint is *triggered*. For this, A is in it's triggering state. Consequently, the context C is restrained from being *completed* until B is *completed*, as well.

A revert of the constraint state from *triggered* back into *initialized* could now be automatically performed when the triggering condition for the constraint no longer holds true.

Assume that feature A is canceled. Work is no longer required to be performed on A or B. A is put into entity state *canceled* and, by this, no longer in a triggering state. The

constraint therefore automatically resets to constraint state *initialized*. This also has the restraints on C lifted as B is no longer required to reach a satisfying state.

Case 2: Reverting from Satisfied to Triggered For the next case, the initial situation has the constraint being *satisfied*. Regarding the example, assume `B` was completed after `A`, and `C` is unrestrained, again.

A revert of the constraint state from *satisfied* back into *triggered* could now be automatically performed for the following reasons:

Cause 1 The satisfying condition for the constraint no longer holds true. The constraint is reverted to *triggered*.

Cause 2 Some CE is changed into a triggering state (again). The constraint is *re-triggered*.

Regarding cause 2, an interesting question is, if the constraint was *re-triggered* by the same entity than before or by a different one. The latter could happen with branched constraints (for examples, see the catalog of constraints in Section 3.3), which may have more than one CE A and more than one CE B. A differentiation into two cases 2A and 2B can be made for cause 2.

Case 2A: Re-Triggered by Same Entity Being *re-triggered* by the same entity is always the case for simple constraints like the Simple Response Constraint (SRespCstr) above. With only one CE A and B, it would always be exactly the same entity leading to the *re-triggering* of the constraint.

Case 2B: Re-Triggered by Different Entity With branched constraints, the possibility of having two CEs `A1` and `A2` comes into play. Note that for the type of branched constraints supported in this concept, the constraint state would change if any single entity out of the set is put into a goal state.

The initial situation is as follows and depicted in Figure 3.7. Assume the constraint was first *triggered* by `A1` and subsequently *satisfied* by `B`. The restraint on `C` is therefore lifted and afterwards, `C` is *completed*.

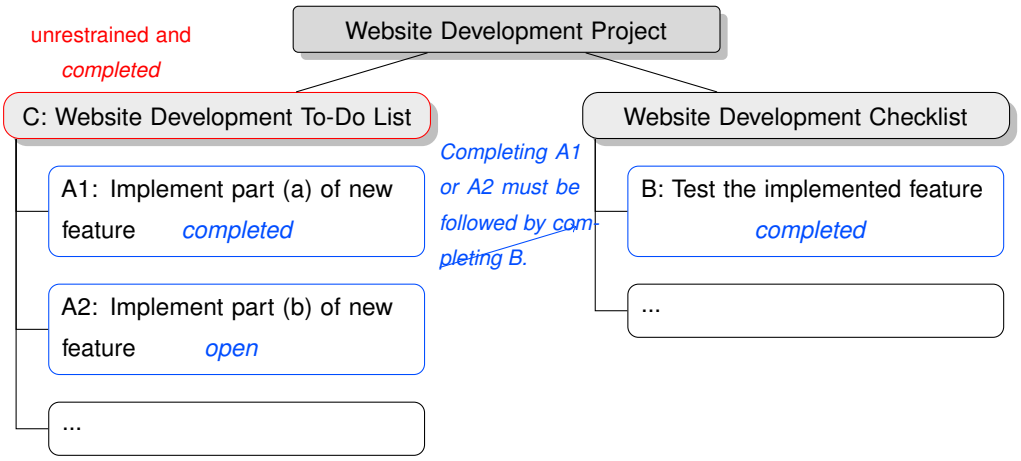


Figure 3.7: Re-triggering a branched constraint: initial situation.

As a next step, consider `A2` reaching it's triggering state (`A2`: *completed*). Consequently, the constraint reverts to constraint state *triggered*. `C` is restrained again (from being *completed*), while already in entity state *completed*. This situation is seen in Figure 3.8.

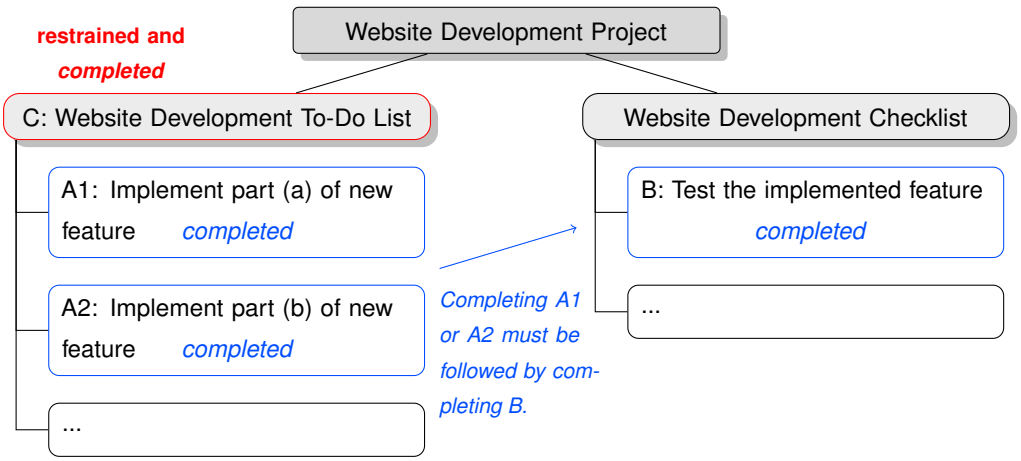


Figure 3.8: Re-triggering a branched constraint: situation after re-triggering.

One could argue that C now being restrained from *completing* while it already is *completed* at the same time is contradicting. On the other hand though, constraints are meant to offer guidance and propose certain behavior with their restraints instead of imposing fixed restrictions. The proposal not to put C into entity state *completed* could still be followed by moving C out of it's current *completed* state. Otherwise, the system could notify the users about the restraint on C so that they still can base their decisions on this information.

The case of having branched constraints re-trigger when a different CE than before reaches its triggering state might be a useful addition to standard constraint state progression, based on how the constraint is expected to act.

Case 3: Resetting from Satisfied Finally, the last case has a *satisfied* constraint being reverted into constraint state *initialized*. This can be caused, when the triggering condition of the constraint is no longer fulfilled.

Consider the previous example scenario with a *satisfied* constraint. When CE A is changed into an entity state different from it's triggering one, this would then lead to the constraint being reset to *initialized*. Restraints that were previously put in place are then consequently lifted.

Possible Problems when Reverting Constraint States Problems can occur, when the reason for an entity's state change that leads to the revert is actually of, e.g., minor importance only, but still influences the constraint's state. Figure 3.9 provides an example, in which a task for creating a requirements documentation (B) can cause a constraint state revert.

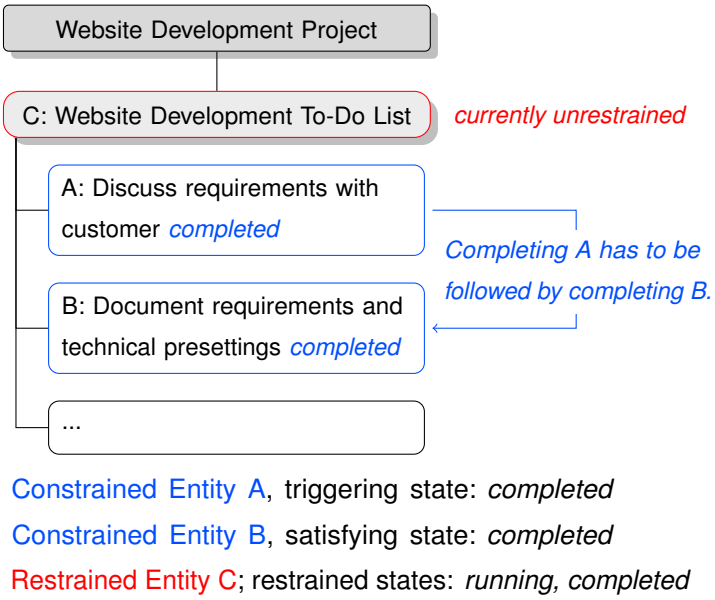


Figure 3.9: Example scenario for an undesired constraint state revert.

As long as the documentation of the requirements (B) remains (*completed*), the constraint is *satisfied* and work on the to-do list (C) (the context of the constraint) is allowed to be performed (the list can be set into state *running*). The constraint state is reverted to *triggered* (see Case 3, Section 3.2.3) as soon as B is no longer in its *satisfying* state. Two reasons for moving the already *completed* task B back into state *running* and therefore out of its *satisfying* state are given below:

Example A An important requirement is currently being added that is of major importance for the future of the project.

Example B The documentation is re-opened for changing a company logo to its updated version and correcting the spelling of the project leaders name.

While example A justifies the *re-triggering* of the constraint and the subsequent restraint on performing work on C, example B does not. The latter case describes minor changes that should actually not lead to set up a restraint for C as this would propose to stop the work on C.

Constraint state reverts are automated and only based on entities' states. It is therefore not easily possible to include the reasoning behind the revert-causing state change of an entity in the rules that would revert the constraints' states.

Solutions to Constraint State Reverts

Cases of constraint state changes deriving from the standard path of progressing through constraint states might lead to implications. The then possible consequences are:

- Undesired re-employing of restraints, which have previously been already lifted.
- Undesired lifting of restraints and thus allowing users (read: proposing that it would be allowed) to perform certain entity state changes .

A couple of proposed solutions for solving the problems that come with constraint state reverts and for improving the functionality for such reverts are shown below. An especially important one is solution 4 (Section 3.2.3), as this one highlights the advantages of including users into the decision making.

Solution 1: Use Manual Constraint State Changes Instead Of Automatic Reverts Automatic reverts of constraint states are a feature that is not a necessity for constraints to fulfill their purpose. Instead of facing complications one could therefore decide to refrain from it completely and let users make use of manual constraint state changes. While it provides less automation, the complexity is kept to a minimum and adjustments are still possible when explicitly chosen. Note that allowing for manual constraint state changes is considered to be an always useful. Regarding usability and frequent changes it might even be a necessary addition.

Solution 2: Make Constraint State Reverts Optional For different cases and scenarios, users could prefer different behavior. One solution therefore would be to implement an option to enable or disable each case of constraint state changes for each specific constraint, individually. The options could be set at constraint creation and changed if needed at any time afterwards.

Solution 3: Work with Refined Entity States While some entity state changes are meant to cause a revert, those of minor relevance are often not. Working with refined entity states could circumvent this problem. The proCollab system, as an example, allows to refine entity states, so that sub-states of an entity state can be created. In the example from earlier (see Section 3.2.3), when an entity is in a sub-state like, e.g., *completed and seeing minor corrections* it would still be considered to be *completed* as well. Thus, the entity would still be in satisfying stateMinor changes could then be done without causing a revert of the constraint's state.

Solution 4: Ask the User This solution can be described as the probably most important one out of the ones presented. Usability is of high importance and needs might change on the minute. Extending the features on the client's side / user interface can offer advantages for all work done with constraints in the system. This proposed solution sees the user being made aware of what an entity state change would lead to before performing it.

Whenever a user is about to change the state of a SE, the system lists all subsequent consequences before performing the change. This includes at least informing of all resulting constraint state changes, the then-placed and then-lifted restraints. In addition, the resulting demands on SEs' for reaching goal states could be presented to the user as well. After having the chance to evaluate these consequences, the user can then make the final decision on either performing the entity state change or choosing against it.

3.2.4 Facts and Variables

All constraints of a same type including this type's simple and branched versions, e.g., the Simple Precedence Constraint (SPrecCstr) and the Branched Precedence Constraint (BrPrecCstr), share the same set of rules. For differentiating between several concrete constraints, facts are the distinguishing factor. Variables for a constraint are presented through facts.

Constraints do not bind the system to supply them with specific entities, types of entities or states as variables. Basically, every SE with an id and every available state can be selected as being a variable (or part of a variable) for constraints. The functionality of the constraints is chosen to be very open, so adjustments and closer restrictions can be made on the side of the system by presenting users with only a specific selection of entities to choose from. This way, the system can draw a smaller circle of options based on it's exact needs.

As the focus was previously on the constraint states, which are part of the rules' conditions, this section discusses the variables and the facts that close the remaining gaps in the constraints' logic. First, an overview of the various basic fact types is given. After that, their interplay and role in the rules' conditions is explained.

Fact Types

Facts are used for presenting various information to the rules. Among others, the basic roles of facts are to present

- the SEs, especially work-items, of the information system in their current states;
- the variables for a constraint; and
- the constraint state.

While entity facts are kept up-to-date through utilizing an entity listener/interceptor, the latter two categories are created and inserted into the rule engine's working memory during a constraint's creation.

Figure 3.10 gives an overview of the different basic fact types representing the constraint state, the variables for constraints and the entity facts.

First, the constraint state is represented by different refined types of constraint state facts: The SRespCstr and the Branched Response Constraint (BrRespCstr) are for example both represented through facts of type *Response Constraint State Fact*. The example shows that simple and branched versions of the same constraint type share the same

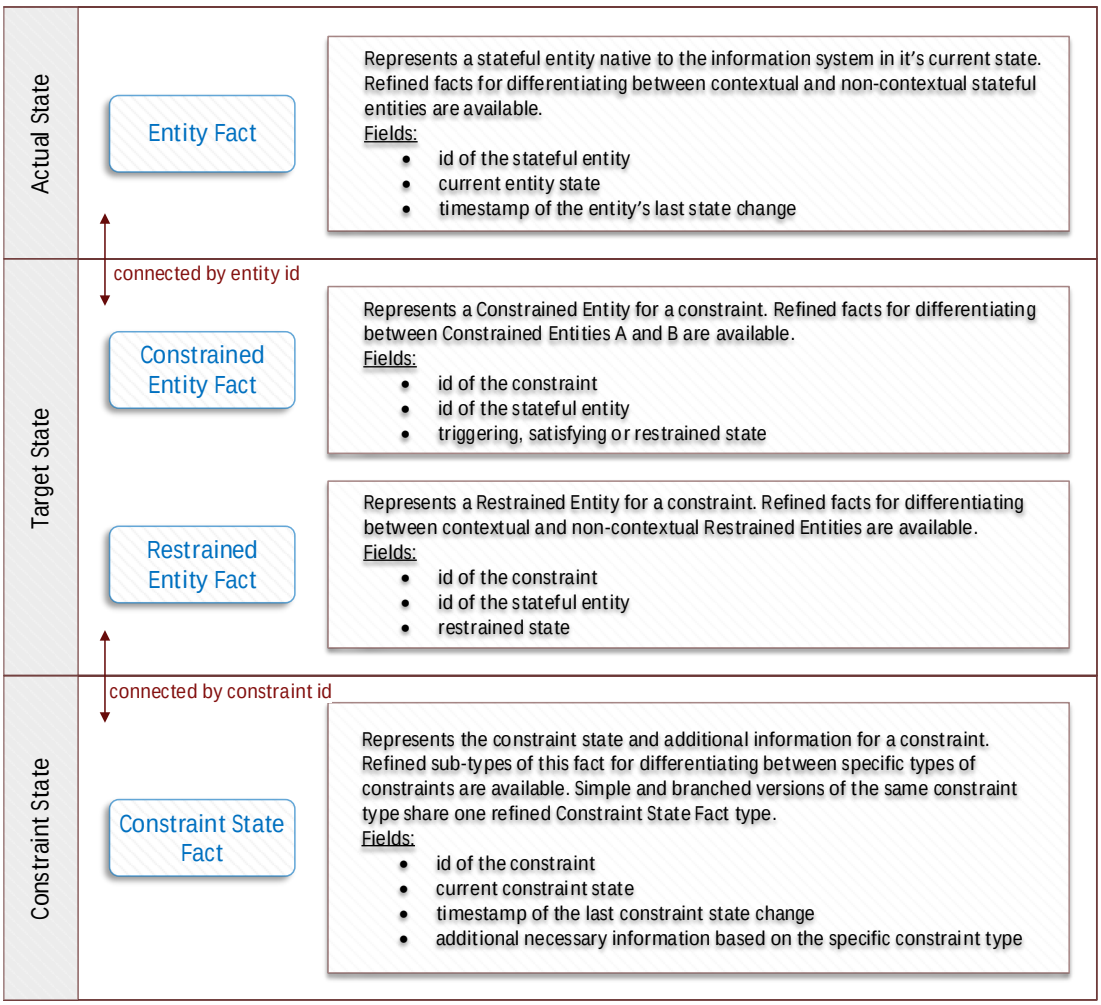


Figure 3.10: Basic fact types used for constraints.

type of Constraint State Fact. The Immediate Response Constraint (ImmRespCstr) on the other hand has its own fact type: the *Immediate Response State Fact*.

Second, Constrained Entity Facts (CEFs) are used to represent the CEs a constraint should work on. They are connected to the corresponding Constraint State Fact (CSF) for this constraint by both referencing the constraint's id. Because of the reference, in case that a CSF signals that a constraint was triggered, a rule can then determine which CEFs belong to this constraint and therefore have to be checked.

In addition, the REs for a constraint are represented via Restrained Entity Facts (REFs). These are connected to the corresponding CSF for this constraint by both referencing the constraint's id.

Next, entity facts as introduced in Section 3.1.3 portray the system's native SEs in their current state. They are connected to the facts for variables above by both referencing the SE's id.

Variables and Facts in Rules

Facts representing the constraint state are used together with those filling the purpose of variables and the entity facts in the LHS of the rules for constraints. The goal is usually either to evaluate if an entity has reached a goal state or to identify an entity that is to be restrained.

For the case of deciding whether to change a constraint state a short example is given. Consider a SRespCstr (see Section 3.3.2) that triggers when CE A reaches one of it's triggering states. Based on the fact types introduced in Section 3.2.4 the a comparison is done in the condition of the rule deciding whether the constraint is to be changed into constraint state triggered. Listing 3.2 presents this comparison written in pseudo code. Note that the rule fires once for each matching combination of facts that was found. E.g., with 2 matching objects of each fact type, the rule in Listing 3.2 does fire $2 \cdot 2 \cdot 2 = 8$ times.

```
1  rule "response_constraint_triggered"
2      when
3          // the $ symbols highlight fields that are stored for the comparison with
               fields of other facts
4          if exists Response Constraint State Fact ( with constraintState == initialized,
5              $cId : constraintId ) AND
6          if exists Constrained Entity A Fact ( with constraintId == $cId,
7              $entId : entityId, $trigState : triggeringState ) AND
8          if exists Entity Fact ( with entityId == $entId and
9              currentState == $trigState  )
10     then
11         change state of constraint with id == $cId to triggered
12 end
```

Code Snippet 3.2: Example rule determining whether a constraint was triggered (pseudo code).

A similar order of comparisons is found in the condition of rules checking for things like, e.g., the sub-entities to be restrained residing in a specific context. Aside of the CSF various other specialized fact types are utilized then. A more detailed example showing the complete set of rules for a constraint can be seen in Section 4.3.2.

3.2.5 Restraints

While facts for variables provide the input for the constraints' conditions, their effects are either setting up or revoking restraints. Figure 3.11 visualized the way restraints are created and checked for. For identifying entities to be restrained, a method very similar to the use of facts for checking if a constraint's state is to be changed is used (see Section 3.2.4). For a more detailed example of a rule please see the example implementation for the ImmRespCstr in Section 4.3.2.

The work with restraints is performed as follows:

- If a constraint state that asks for setting up restraints is reached, the entities to be restrained are identified via the available facts.
- The rule calls an external Java method for setting up the restraint.
- A restraint object is created and persisted on the Java side.

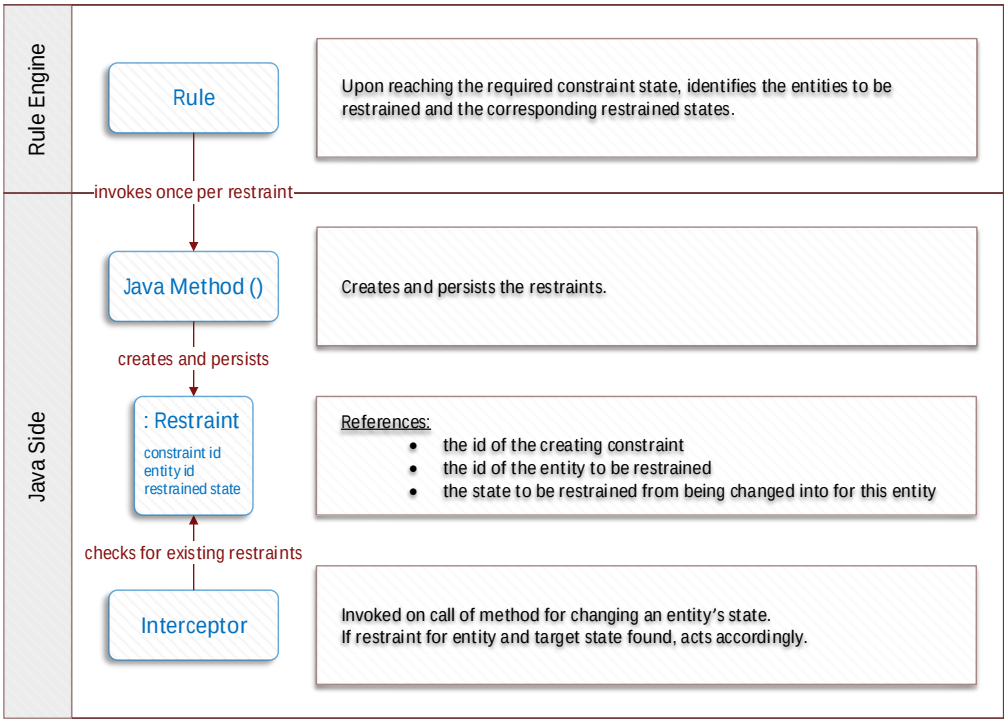


Figure 3.11: Creating and checking for restraints.

- The system can check for these objects and the referenced REs and restrained states to decide if a state change should be done and act as needed.
- The removal of restraints is done by removing all persisted restraints that reference a specific constraint id.

For each state that is meant to be restrained there is one restraint object created. This work refers to each of these objects as a (one) restraint. The way restraints are realized by this is mirroring the work of the rule engine on facts representing small, atomic objects like, e.g., single elements of a collection. The information held by a restraint is comprised of

- the id of the creating constraint,
- the id of the entity this restraint is meant for, and
- one state of the entity that is to be restrained.

The check for existing restraints by the system can then, e.g., be performed right when an entity's state is about to be changed. The implementation, for example, uses an interceptor to check for restraints when the method for changing an entity's state is called (see Section 4.2.1).

3.2.6 Working With Constraints

In the following, a brief step-by-step summary of how constraints are worked with is shown.

The work begins with a user creating a new constraint:

- The variable entities and their states are chosen for the constraint.
  - During the creation, the user selects the entity A and usually entity B for this constraint by the entity's ids.
  - For each chosen entity A and B, one or more triggering states, satisfying states and/or restrained states are selected depending on the constraint's type.

  - In addition, some constraints have the user choose one or multiple CSEs to define the context(s) over which this constraint's restraining effect is to span (again by entity's ids).
  - For each of the chosen CSEs one or more restrained states are selected.
  - If the constraint automatically selects any further CSEs to be restrained (sub-entities of a context), one or more restrained states shared among these have to be chosen.

- Facts representing the variable parts and the constraint state (set to *initialized*) are inserted into the rule engine's working memory.

On side of the rule engine, rules are evaluated whenever the facts for them do change. As soon as the ones for this constraint are available to the rule engine its rules are being evaluated as well

- Facts representing the variable entities are compared to facts representing the system's native SEs. The current constraint state is taken into consideration as well, thus determining which rules can fire. Depending on the outcome of the evaluations the constraint state is changed accordingly.
- Based on the constraint state, further rules for setting up or revoking restraints are fired. Restraints are created as persisted objects on Java side. Each of them references the creating constraint, an entity and a state this entity is restrained from changing into.

The system checks for existing (persisted) restraints every time a state change for an entity is about to be performed. When a user tries to do so the system goes through the persisted restraints looking for one that applies to the entity and target state in question. When a restraint is found the system can react to it as needed, with the former therefore offering guidance in form of proposals for certain behavior, instead of strict ruling.

3.2.7 Summary

The different theoretical components or constructs that make up the concept for constraints and their interplay were discussed in the preceding sections. The key points are recapped below:

- A rule engine is used as the driving part of technology.
- The concept common for all constraint types is detailed part for part.
- Constraint states are used to decide which rules are allowed to fire.
- Aside of the standard path of progressing through the constraint states several cases for reverting states are described together with accompanying complications.
- Several solutions for best handling constraint state changes are presented.
- The use of facts for representing the constraints' variables is explained.
- And finally, the restraints, represented by objects on the traditional code's side, are shortly laid out.

3.3 Catalog of Selected Constraints

A number of constraints was identified to be especially useful in a high number of situations. The selection was done based on the existing use cases for the proCollab system and orients itself on the characteristics of KiPs. In addition, the related work on [23], [25], [26], [35], [36] was influential and helpful for selecting the constraints, categorizing them, defining their behavior, phrasing their semantic meanings and visualizing each one. A catalog containing these constraints separated into different categories is presented in this section. Additional types can be found helpful and be based off the selection presented here. The following constraints were therefore also chosen for providing examples and form a very solid baseline for further extension.

The branched constraints presented in this section do all change their states as soon as *any* of the CEs in their sets of CEs reaches a goal state. This means that any single

entity of a set is enough to lead to the constraint changing it's state. In comparison, it is also possible to define branched constraints so that the constraint state is changed only when *all* the CEs of a set reach a goal state. This additional version of branched constraints is not supported by the concept and worth considering as a future extension.

The figures for representing the constraints' functionality in this section follow the concept for constraints and its representation introduced in Section 3.2. All use cases shown for the categories of constraints are based on the website development use case presented in Section 2.3.

3.3.1 Precedence Constraints

The Precedence Constraints describe a relationship between SEs that requires one or more SEs to be in a certain goal state before others could change into their restrained states. This section describes the concepts for the different types of Precedence Constraints that were found to be practical in likely use cases.

Figure 3.12 provides an example use case for precedence constraints: Before a website can be released (A), it first must be discussed with the customer (B). While the discussion can be performed without any subsequent releases following it, the other way around requires the discussion preceding the website being published.

Figure 3.12: A Precedence Constraint in the website development use case.

The Simple Precedence Constraint

For the SPrecCstr, there is a dependency between two SEs `SE_A` and `SE_B` requiring that for `SE_B` to be set into a restrained state, `SE_A` must have been set into a satisfying state at any time before.

A representation of the SPrecCstr's concept is given in Figure 3.13.

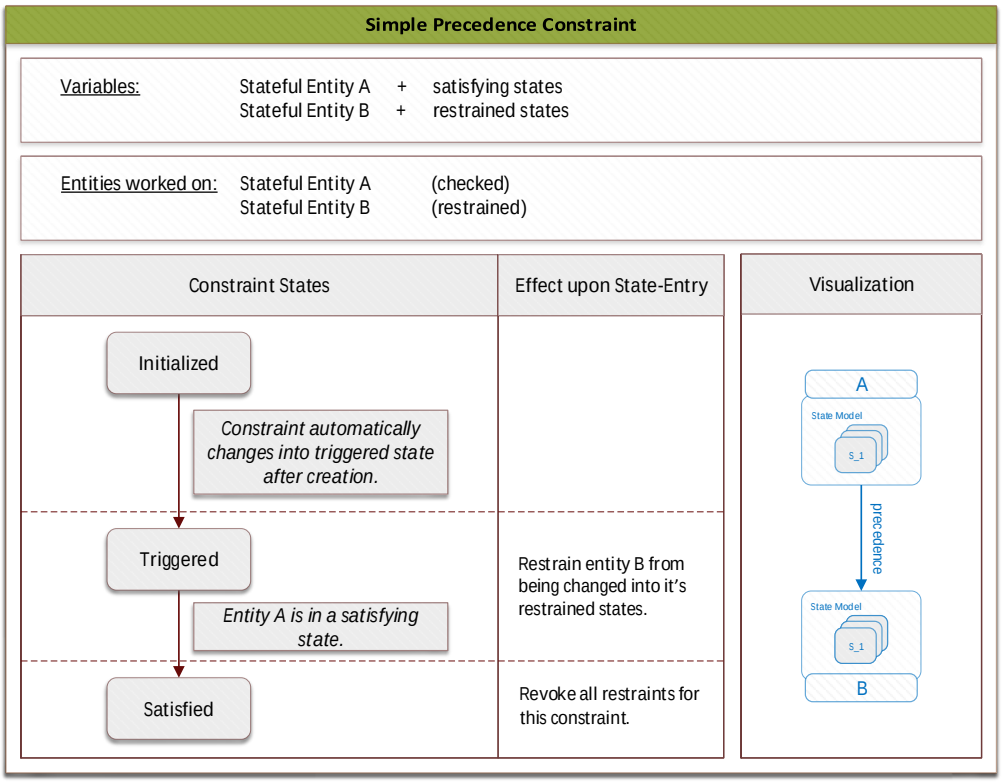


Figure 3.13: Concept for the Simple Precedence Constraint.

The Branched Precedence Constraint

For the BrPrecCstr, there is a dependency between two sets of SEs `SES_A` and `SES_B` requiring that for any SE from `SES_B` to be set into a restrained state, any SE from `SES_A` must have been set in a goal state at any time before.

A representation of the BrPrecCstr's concept is given in Figure 3.14.

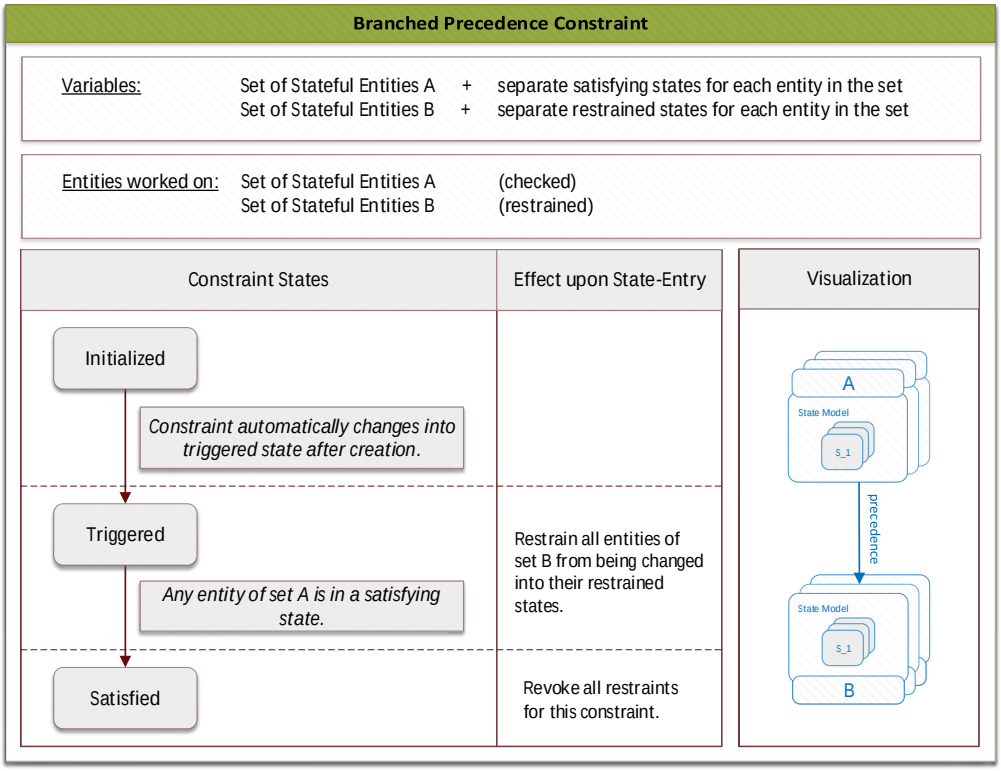


Figure 3.14: Concept for the Branched Precedence Constraint.

The Immediate Precedence Constraint

The ImmRespCstr is an example for an impractical constraint and mentioned here for the sake of completness and to underline the difficulties coming with it.

Similar to the SPrecCstr, the ImmRespCstr expresses a dependency between two SEs `SE_A` and `SE_B`. The important difference lies in it's immediacy. It requires that for `SE_B` to be set into a restrained state, `SE_A` must have been set into a satisfying state immediately before. This means, no other entity must have been changed into a state defined to be relevant in between A and B.

This constraint does not restrain any state changes for entities other than `SE_B`. Changing `SE_B` into any restrained state is not a must, and the ImmRespCstr only describes a requirement for doing so. Therefore, it is possible that after `SE_A` was changed into a satisfying state, some state change is performed for any other entity (different from

SE_B). This would interrupt the immediacy and render the constraint no longer satisfiable. Satisfying the ImmRespCstr is therefore mostly depending on luck.

3.3.2 Response Constraints

Response Constraints express a relationship between SEs that requires that, after certain state changes were done for one or more of them, a goal state is reached by another. This section describes the concepts for the different types of Response Constraints that were found to be practical in likely use cases.

Figure 3.15 provides an example use case for precedence constraints: After the draft for a website was updated (A), it has to be approved by the customer (B). B can change it's state without any further consequences. After A was moved into any of it's triggering states, though, the to-do list C (the context of A and B) will be restrained from completing until B has reached a satisfying state. The context of A and B (being C) should be restrained from being completed as long as A is completed but B is still not.

Figure 3.15: A Response Constraint in the website development use case.

The Simple Response Constraint

For the SRespCstr, there is a dependency between two SEs SE_A and SE_B requiring that when SE_A is set into a triggering state (triggering the constraint), SE_B must be

set into a satisfying state at any time afterwards (satisfying the constraint). When the constraint is triggered, the entities of a set of one or more specified CSEs `CSES_C` will be restrained from being changed into their restrained states, until the constraint is satisfied.

A representation of the SRespCstr's concept is given in Figure 3.16.

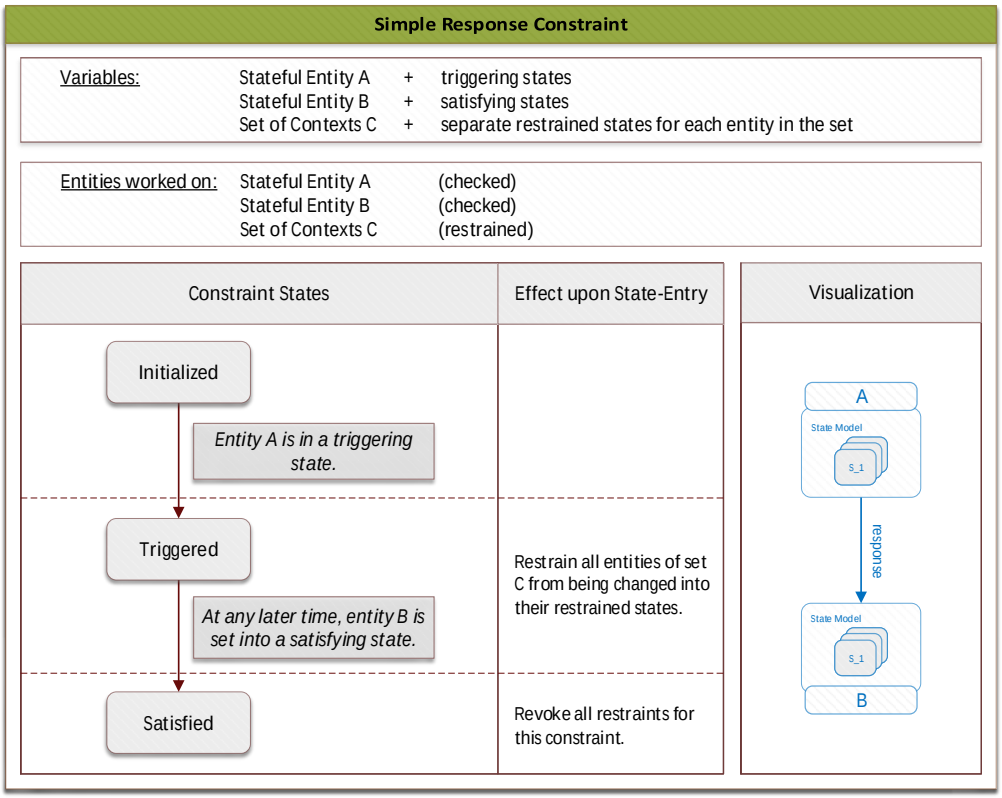


Figure 3.16: Concept for the Simple Response Constraint.

The Branched Response Constraint

For the BrRespCstr, there is a dependency between two sets of SEs `SES_A` and `SES_B` requiring that when any SE from `SES_A` is set into a triggering state (triggering the constraint), any SE from `SES_B` must be set into a satisfying state at any time afterwards (satisfying the constraint). When the constraint is triggered, the entities of a set of one or more specified CSEs `CSES_C` will be restrained from being changed into their restrained states, until the constraint is satisfied.

A representation of the BrRespCstr's concept is given in Figure 3.17.

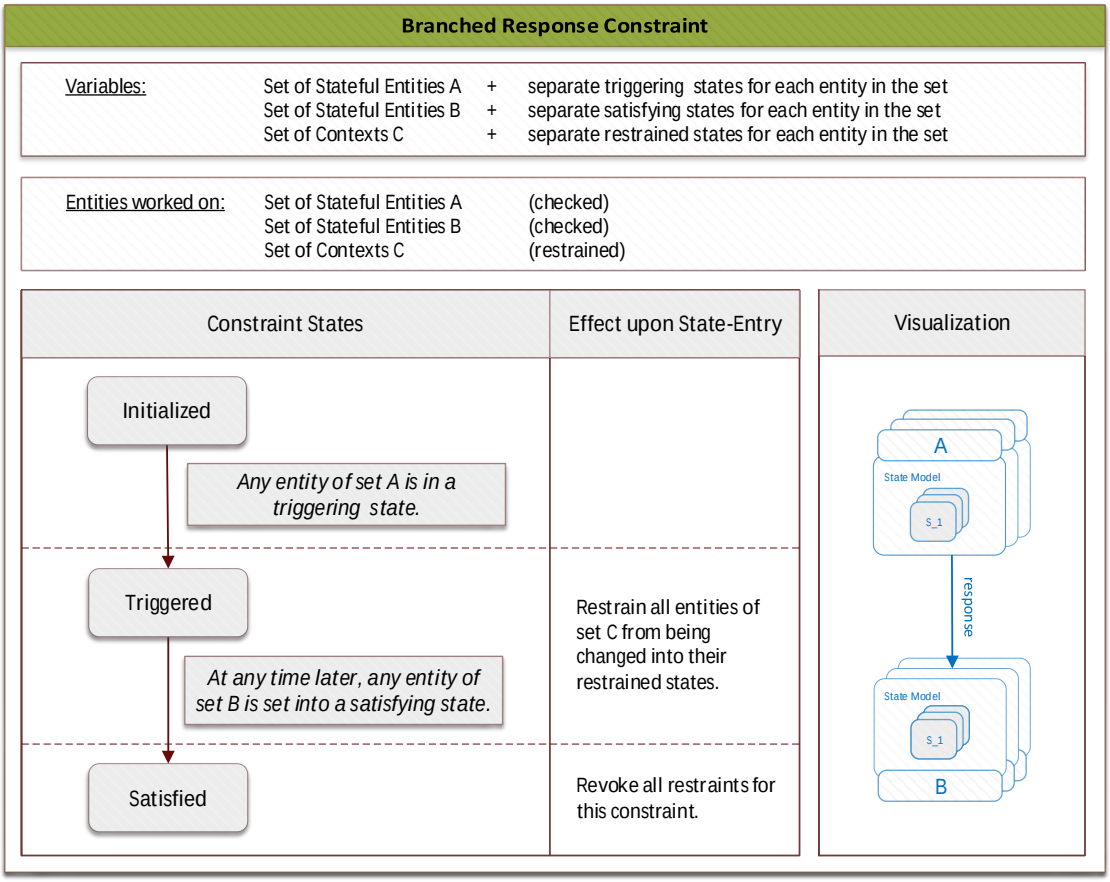


Figure 3.17: Concept for the Branched Response Constraint.

The Immediate Response Constraint

For the ImmRespCstr, there is a dependency between two SEs `SE_A` and `SE_B` requiring that when `SE_A` is set into a triggering state (triggering the constraint), `SE_B` must be set into a satisfying state immediately afterwards (satisfying the constraint). When the constraint is triggered, the entities of a set of one or more specified CSEs `CSES_C` and all their automatically identified stateful sub-entities will be restrained from being changed into their restrained states, until the constraint is satisfied.

A representation of the ImmRespCstr's concept is given in Figure 3.18.

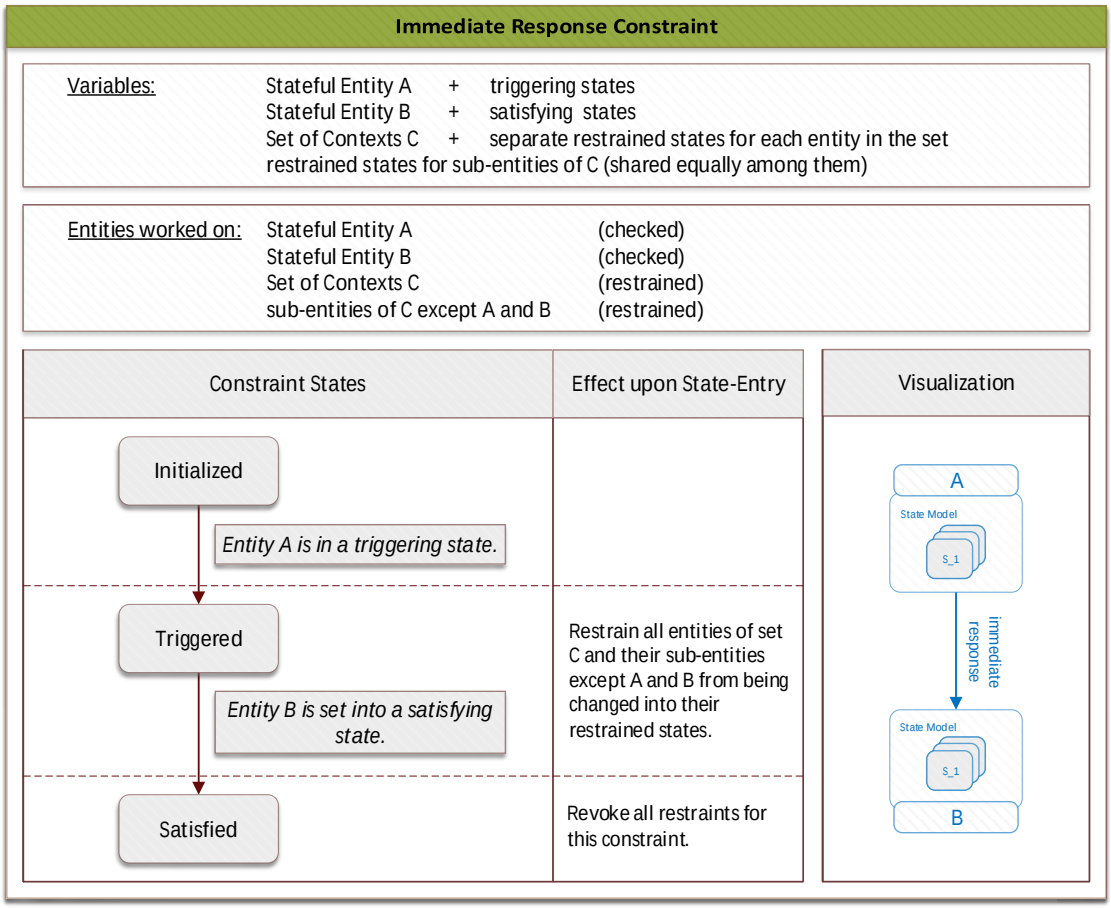


Figure 3.18: Concept for the Immediate Response Constraint.

3.3.3 Succession Constraints

Succession Constraints form a combination of Precedence and Response Constraints. They express a relationship between SEs that requires either both of them to be changed into specific states in order, or none of them see the selected state changes at all. This section describes the concepts for the different types of Succession Constraints that were found to be practical in likely use cases.

An example is presented in Figure 3.19. There, the documentation of the requirements can only be checked for being complete (meaning it contains every important requirement) (`B`), after the creation of the documentation of requirements and technical presettings (`A`) has been completed (meaning finished) at any time earlier. In addition, after the documentation's making was finished (`A`), it has to be checked for being complete (`B`). The contexts of `A` and `B` (being `C1` and `C2`) should be restrained from being changed into state completed as long as `A` is in state completed but `B` is not.

Here, a two-way dependency exists between the two tasks `A` and `B`. In contrast, a Response Constraint would allow `B` to see its state changed without any requirements on it. And a Precedence Constraint has no requirements on state changes done for `A`. The Succession Constraints therefore form a combination of a Response and a Precedence Constraint.

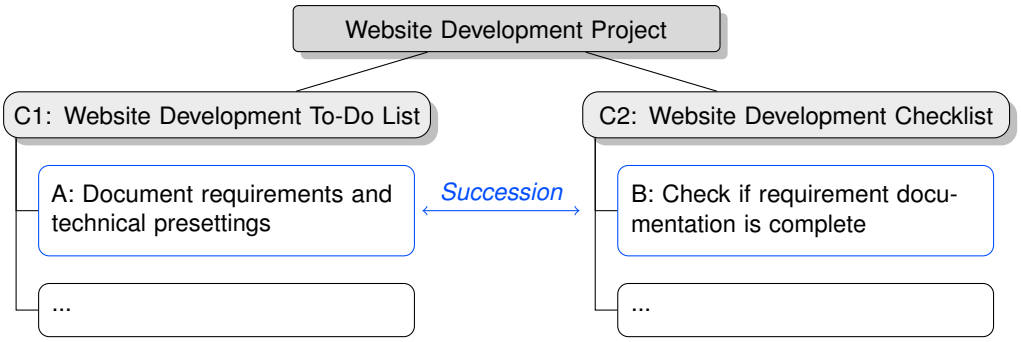


Figure 3.19: A Succession Constraint in the website development use case.

The Simple Succession Constraint

For the SSuccCstr, there is a dependency between two SEs `SE_A` and `SE_B` requiring that for `SE_B` to be set into a restrained state, `SE_A` must have been set into a goal state at any time before (this is the precedence part of the constraint). In addition, there is a dependency between `SE_A` and `SE_B` requiring that when `SE_A` is set into a goal state, `SE_B` must be set into a satisfying state at any time afterwards (this is the response part of the constraint). When the response part is triggered, the entities of a set of one or more specified CSEs `CSES_C` will be restrained from being changed into their restrained states, until the constraint is satisfied. For signaling that the response part is triggered (being equal to the precedence part being satisfied), an additional signaling flag is added to this constraint's CSF.

A representation of the SSuccCstr's concept is given in Figure 3.20.

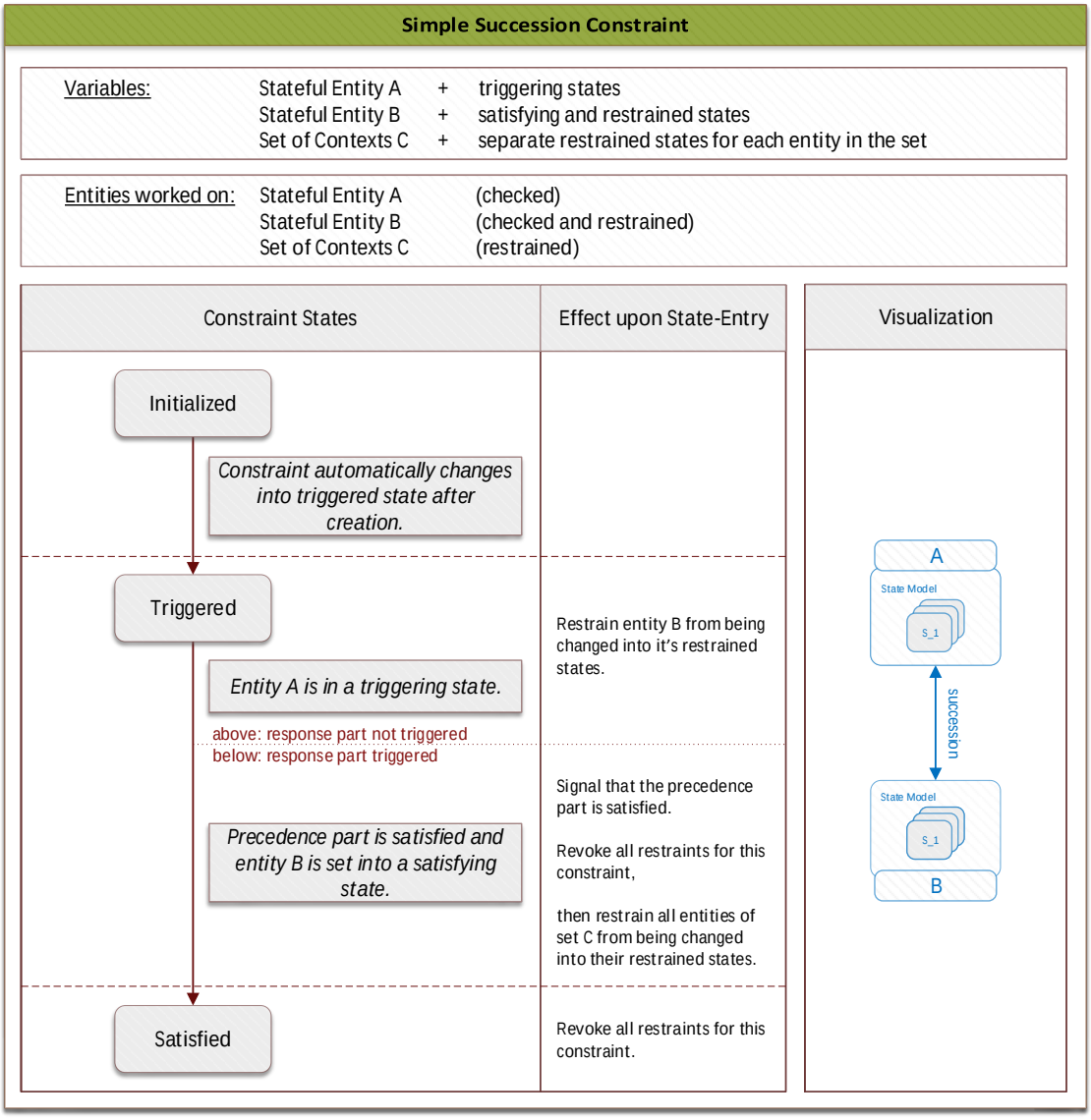


Figure 3.20: Concept for the Simple Succession Constraint.

The Branched Succession Constraint

For the Branched Succession Constraint (BrSuccCstr), there is a dependency between two sets of SEs SES_A and SES_B requiring that for any SE of SES_B to be set into a restrained state, any SE of SES_A must have been set into a goal state at any time before (this is the precedence part of the constraint). In addition, there is a dependency between SES_A and SES_B requiring that when any SE of SES_A is set into a goal state, any SE of SES_B must be set into a satisfying state at any time afterwards (this is the response part of the constraint). When the response part is triggered, the entities of a set of one or more specified CSEs CSES_C will be restrained from being changed into their restrained states, until the constraint is satisfied. For signaling that the response part is triggered (being equal to the precedence part being satisfied), an additional signaling flag is added to this constraint's CSF.

A representation of the BrSuccCstr's concept is given in Figure 3.21.

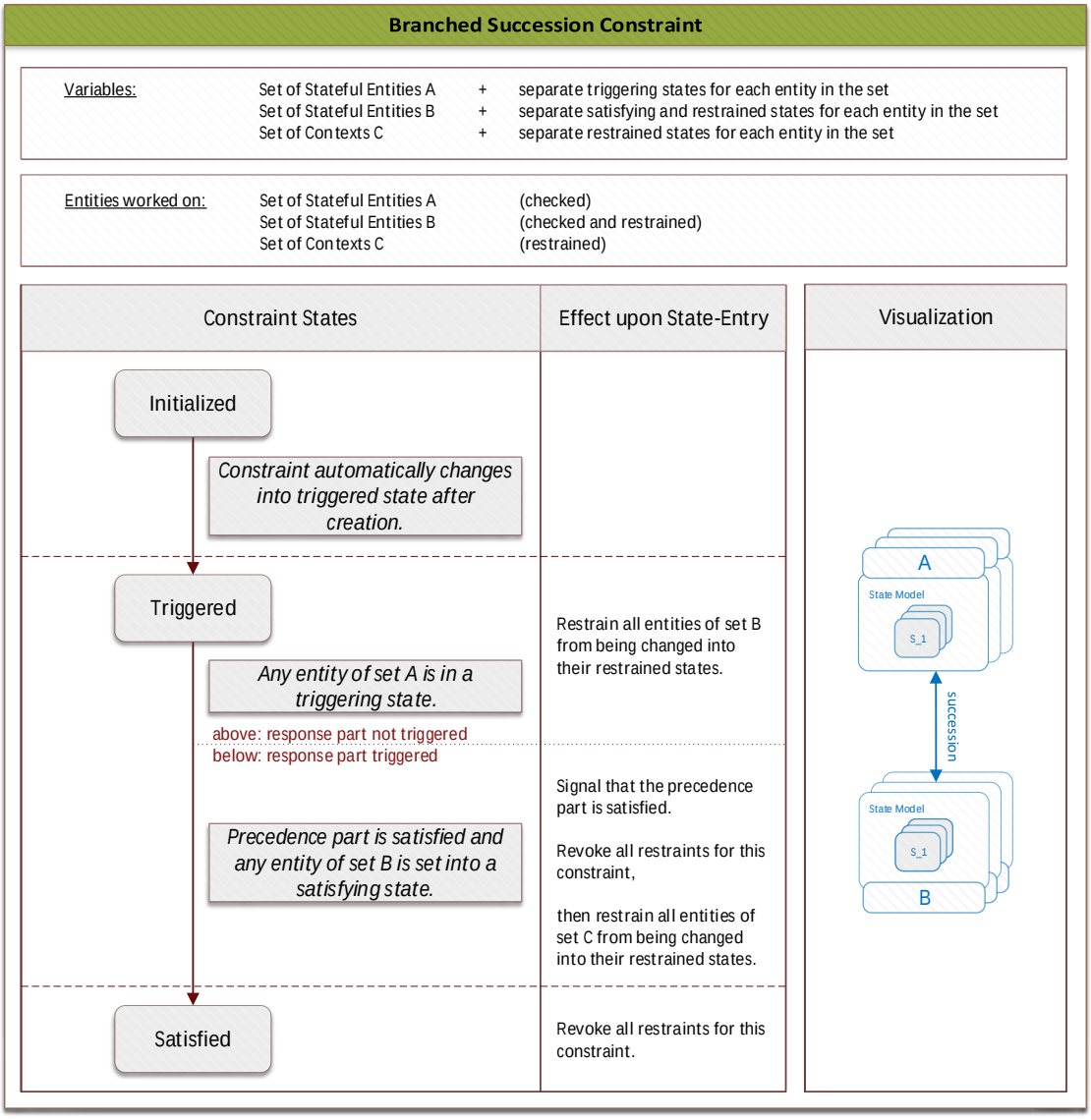


Figure 3.21: Concept for the Branched Succession Constraint.

The Immediate Succession Constraint

For the Immediate Succession Constraint (ImmSuccCstr), there is a dependency between two SEs `SE_A` and `SE_B` requiring that for `SE_B` to be set into a restrained state, `SE_A` must have been set into a goal state immediately before (this is the precedence part of the constraint). In addition, there is a dependency between `SE_A` and `SE_B` requiring that when `SE_A` is set into a goal state, `SE_B` must be set into a satisfying state immediately afterwards (this is the response part of the constraint). When the response part is triggered, the entities of a set of one or more specified CSEs `CSES_C` and all their automatically identified stateful sub-entities will be restrained from being changed into their restrained states, until the constraint is satisfied. By restraining the sub-entities the constraint is preventing them for interrupting. For signaling that the response part is triggered (being equal to the precedence part being satisfied), an additional signaling flag is added to this constraint's CSF.

A representation of the ImmSuccCstr's concept is given in Figure 3.22.

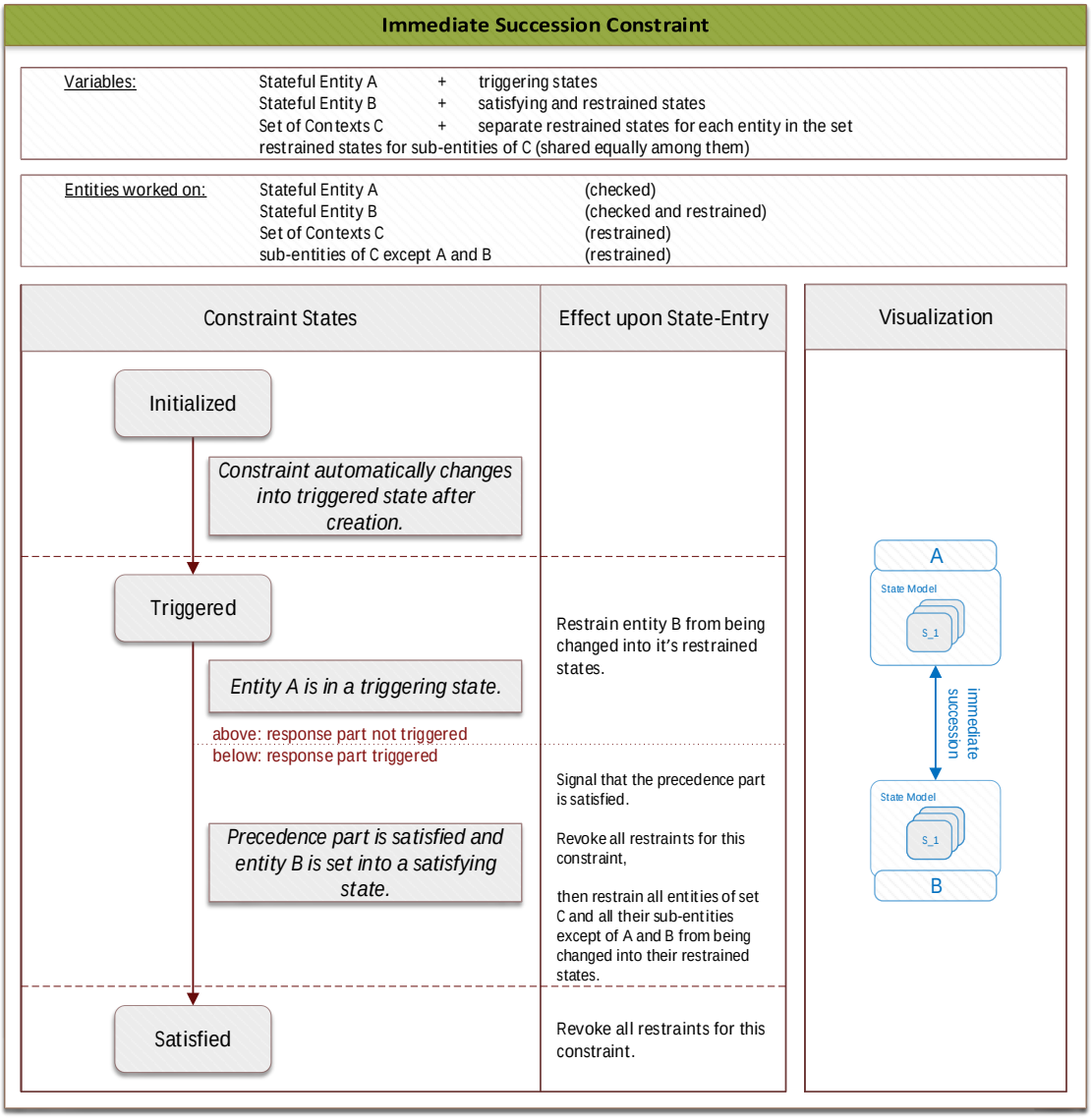


Figure 3.22: Concept for the Immediate Succession Constraint.

3.3.4 Coexistence Constraints

Constraints belonging to this group define combinations of states that the related SEs must reach. There is no specific order required, in which the entities would need to reach one of their goal states. Similar to the Response and Succession Constraints (see Sections 3.3.2 and 3.3.3, respectively), they also affect one or more CSE and restrain it/them from changing into given states as long as the Coexistence Constraint is triggered but not satisfied.

Figure 3.23 depicts an example use case: If the requirement documentation is checked for completeness (B), the requirements' consistency will need to be checked (A) as well, and vice versa. This is done because achieving consistency without completeness or completeness without consistency can be assumed to be of little value. The context of A and B (being C) should be restrained from being completed as long as either A or B is completed, but the remaining one is not.

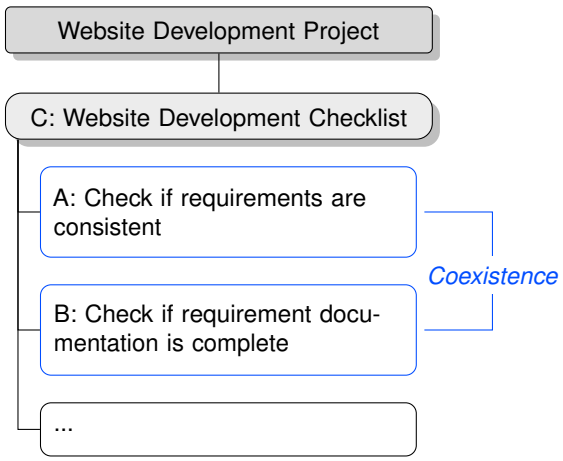


Figure 3.23: A Coexistence Constraint in the website development use case.

The Simple Coexistence Constraint

For the Simple Coexistence Constraint (SCoexCstr), there is a dependency between two stateful entities SE_A and SE_B requiring that when any of them is set into a triggering state (triggering the constraint), the other must be set into a satisfying state at any time

afterwards (satisfying the constraint). When the constraint is triggered, the entities of a set of one or more specified CSEs `CSES_C` will be restrained from being changed into their restrained states, until the constraint is satisfied.

A representation of the SCoexCstr's concept is given in Figure 3.24.

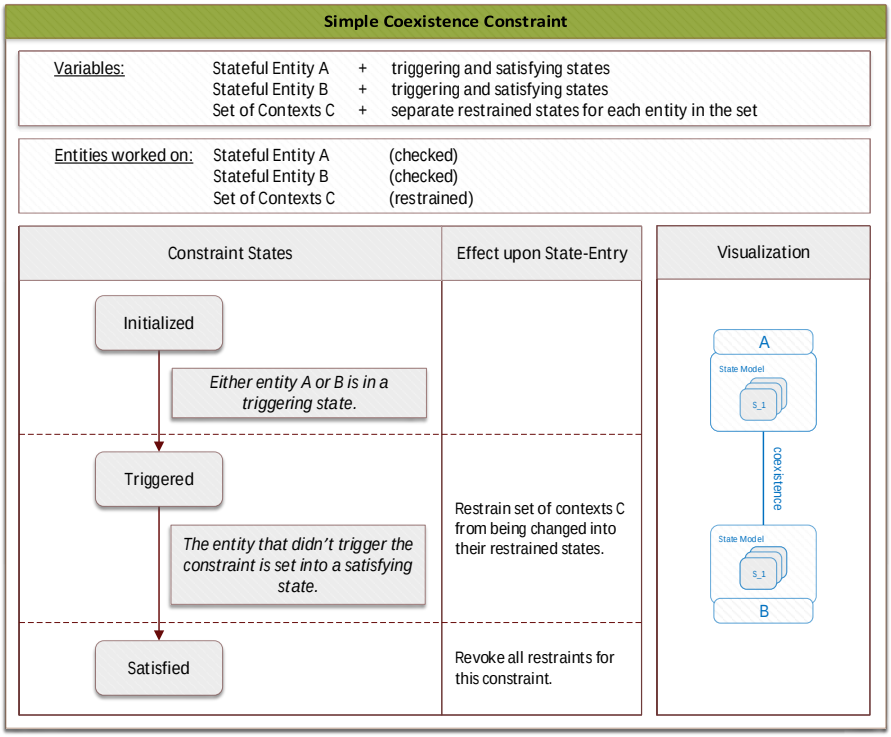


Figure 3.24: Concept for the Simple Coexistence Constraint.

The Branched Coexistence Constraint

For the Branched Coexistence Constraint (BrCoexCstr), there is a dependency between two sets of stateful entities `SES_A` and `SES_B` requiring that when any SE of one of the sets is put into a triggering state (triggering the constraint), any SE from the other set must be put into a satisfying state at any time afterwards (satisfying the constraint). When the constraint is triggered, the entities of a set of one or more specified CSEs `CSES_C` will be restrained from being changed into their restrained states, until the constraint is satisfied.

A representation of the BrCoexCstr's concept is given in Figure 3.25.

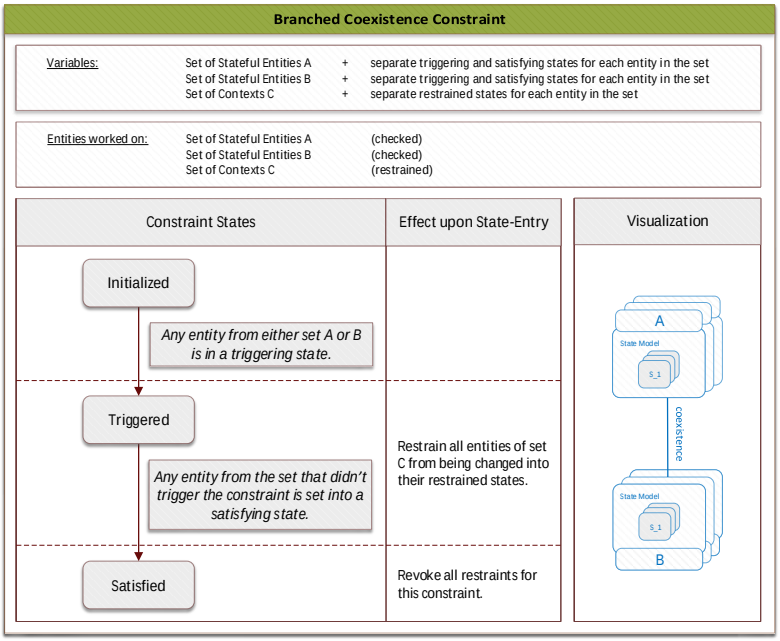


Figure 3.25: Concept for the Branched Coexistence Constraint.

3.3.5 Existence Constraints

Constraints belonging to this group express a relationship between SEs and one or more specified contexts and their automatically identified sub-entities. They define that certain states must be reached by the CEs before other given state changes can be performed by the CSEs and their sub-entities. While for other categories of constraints the effects are often aimed at specified entities and contexts, Existence Constraints do work to a large extend on automatically identified sub-entities.

The use case presented in Figure 3.26 gives an example, where the project is to develop a new website based on an older one. The evaluation of the old website (`A_1`) is the first task that must be completed within the context `C`. Only then work on other sub-tasks (not depicted) of `C` can be started.

Providing another example for a different constraint, publishing the website (A_2) should be the necessary last task to be completed for the to-do list (C). Only after it is finished, the to-do list should be allowed to be declared completed itself.

Figure 3.26: Existence Constraints in a website development use case.

The Init Constraint

For the Init Constraint (InitCstr), a specified SE has to be set into a certain satisfying state before a set of one or more specified CSEs and their automatically identified stateful sub-entities can be changed into their restrained states.

A representation of the InitCstr's concept is given in Figure 3.27.

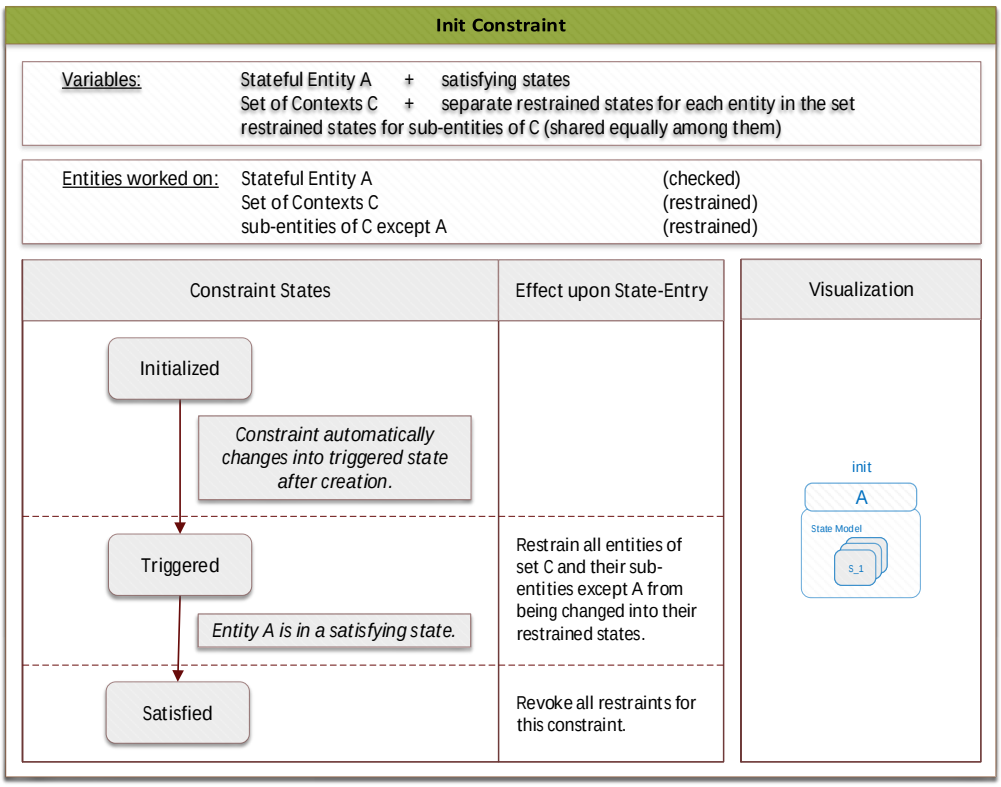


Figure 3.27: Concept for the Init Constraint.

The Completion Constraint

For the ComplCstr, a specified SE `A` has to be set into a certain satisfying state before a set of one or more specified CSEs `CSES_C` can be changed into their restrained states. Furthermore, after `A` was set into a satisfying state, none of the automatically identified stateful sub-entities of `CSES_C` may be set into a restrained state, anymore.

Note, that this constraint could also automatically put the CSEs of `CSES_C` into specific states as soon as it is satisfied. An example would be setting a context to completed when the completing entity `A` reaches a satisfying state. But as this concept designs constraints to work with restraints for provide guidance only, this would lead away from the constraint's responsibilities. An automated action (see Section 3.5.1) could change the state of the context, though, if the task management system wants to do so.

A representation of the ComplCstr's concept is given in Figure 3.28.

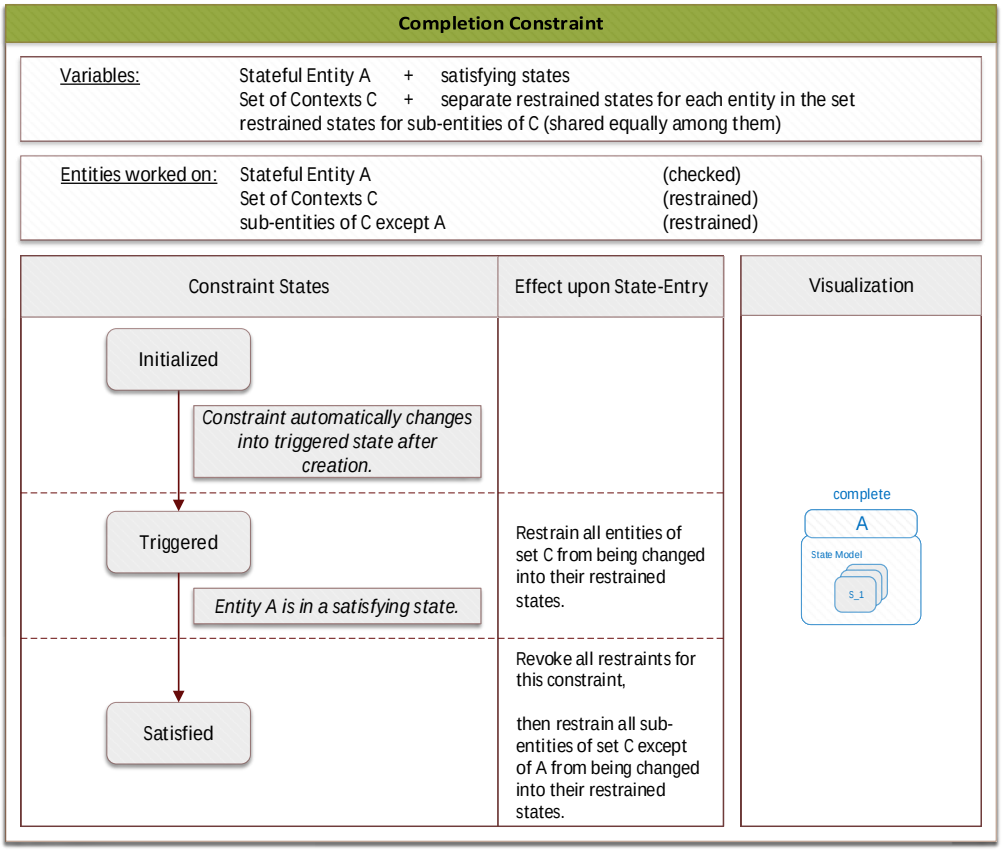


Figure 3.28: Concept for the Completion Constraint.

3.3.6 Negation Constraints

Negation Constraints describe a relationship that leads to the restraining of state changes for one or more SEs as soon as another set of one or more SEs reaches specified states.

Note that the Negated Simple Precedence, Negated Simple Response and Negated Simple Succession Constraints are effectually equal to each other. The same holds true for their branched variants. This concept therefore does only lie out the Simple and Branched Negated Succession Constraints.

The Negated Simple Succession Constraint

The Negated Simple Succession Constraint (NegSSuccCstr) does combine both the Negated Simple Precedence Constraint (NegSPrecCstr) as well as the Negated Simple Response Constraint (NegSRespCstr) into a single constraint.

For the NegSSuccCstr, there is a dependency between two SEs `SE_A` and `SE_B` requiring that when `SE_B` is to be set into a certain restrained state, `SE_A` can not have been set into a specific triggering state at any time before. Reversely, when `SE_A` has been set into a triggering state, `SE_B` can not be set into a restrained state at any time after.

A representation of the NegSSuccCstr's concept is given in Figure 3.29.

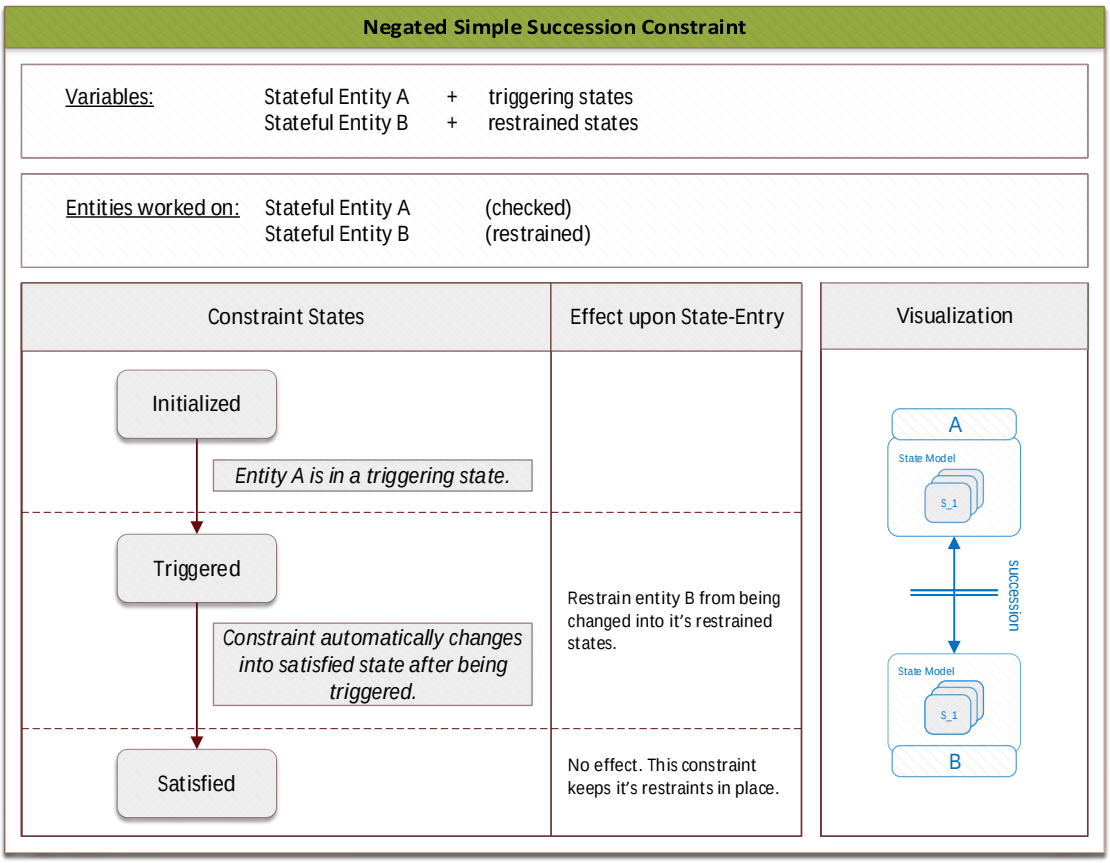


Figure 3.29: Concept for the Negated Simple Succession Constraint.

The Negated Branched Succession Constraint

The Negated Branched Succession Constraint (NegBrSuccCstr) does combine both the Negated Branched Precedence Constraint (NegSPrecCstr) as well as the Negated Branched Response Constraint (NegSRespCstr) into a single constraint.

For the NegBrSuccCstr, there is a dependency between two sets of SEs `SES_A` and `SES_B` requiring that when any SE of `SES_B` is to be set into a certain restrained state, no SE of `SES_A` can not have been set into a specific triggering state at any time before. Reversely, when any SE of `SES_A` has been set into a triggering state, no SE of `SES_B` can be set into a restrained state at any time after.

A representation of the NegBrSuccCstr's concept is given in Figure 3.30.

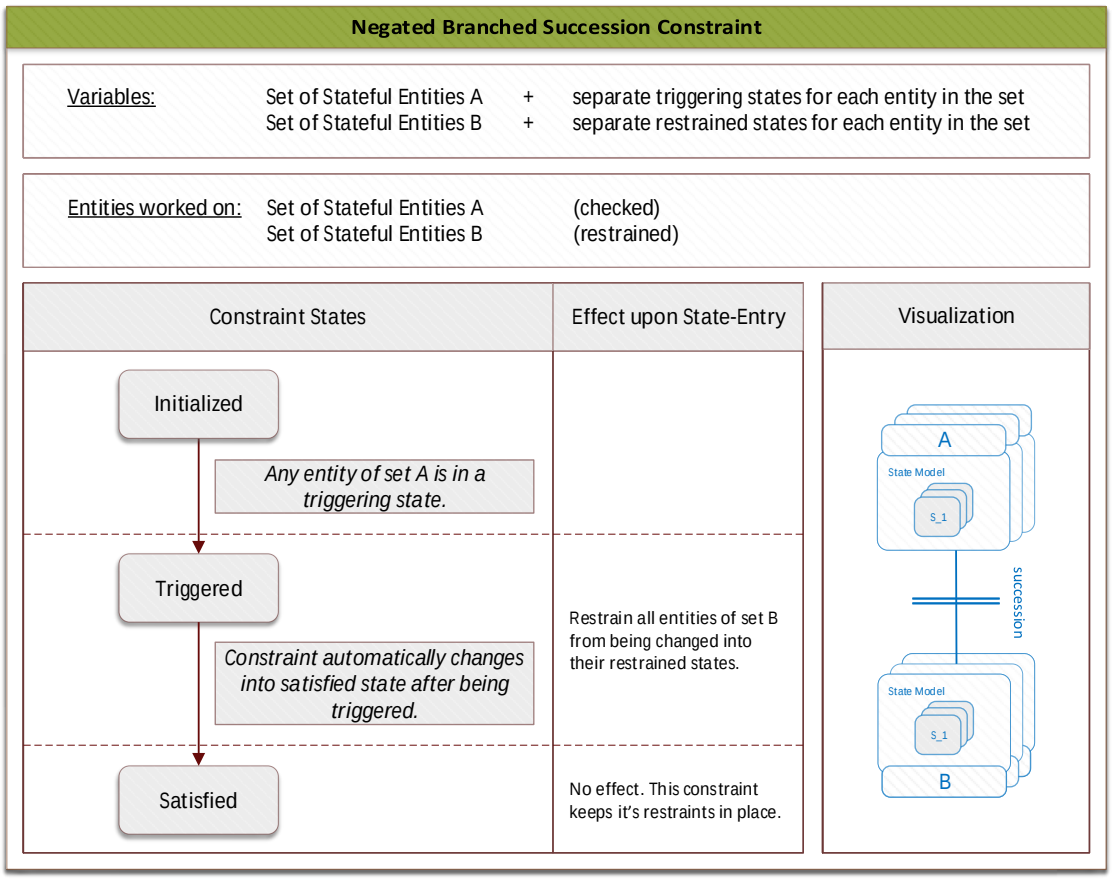


Figure 3.30: Concept for the Negated Branched Succession Constraint.

The Negated Simple Coexistence Constraint

For the Negated Simple Coexistence Constraint (NegSCoexCstr) there is a dependency between two SEs `SE_A` and `SE_B` requiring that when any of them is set into a triggering state, the other may not be set into a restrained state at any time afterwards.

A representation of the NegSCoexCstr's concept is given in Figure 3.31.

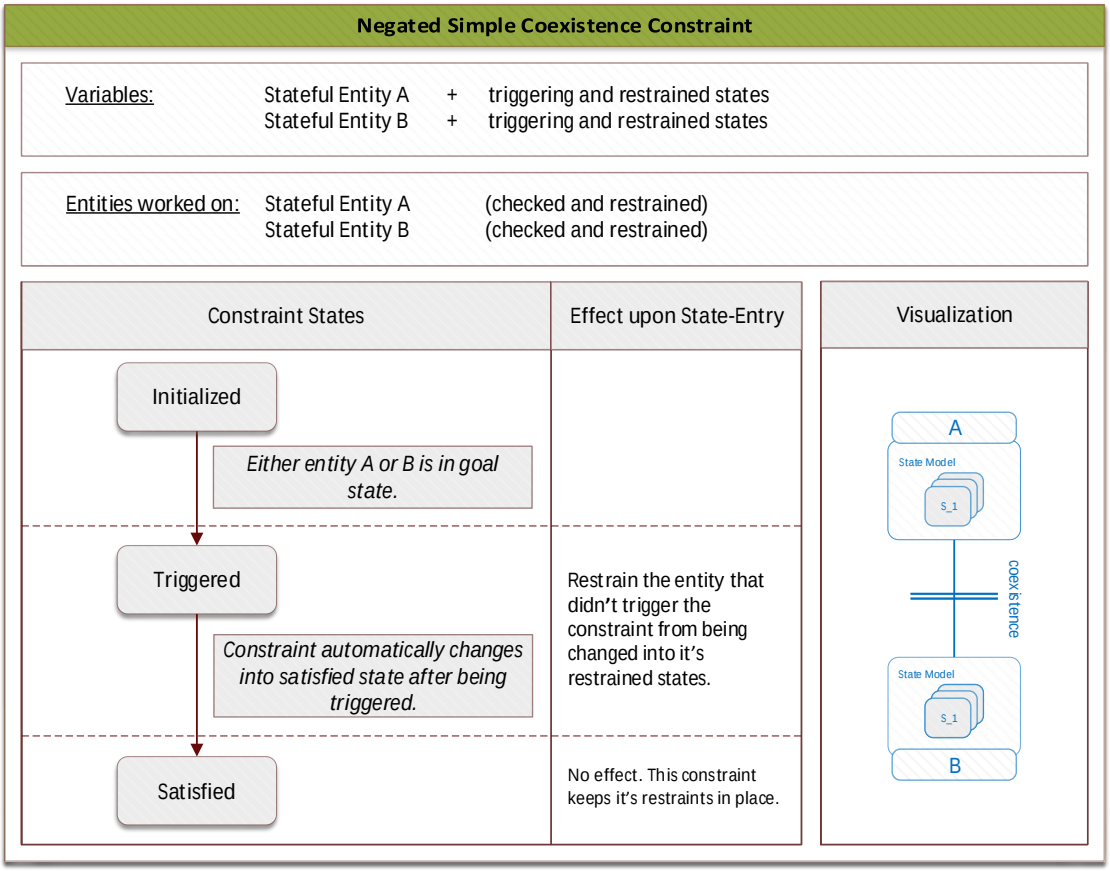


Figure 3.31: Concept for the Negated Coexistence Constraint.

The Negated Branched Coexistence Constraint

For the Negated Branched Coexistence Constraint (NegBrCoexCstr), there is a dependency between two sets of SEs `SES_A` and `SES_B` requiring that when any SE of one

of the sets is put into a triggering state, none of the other set's SEs must be put into a restrained state at any time afterwards.

A representation of the NegBrCoexCstr's concept is given in Figure 3.32.

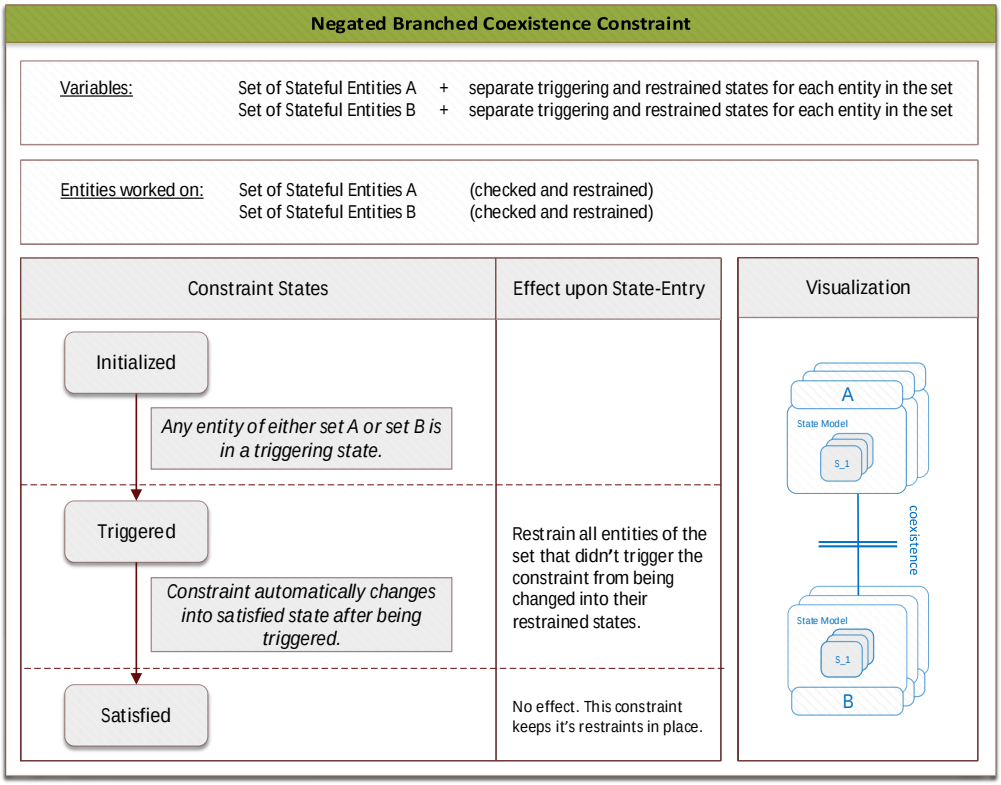


Figure 3.32: Concept for the Negated Branched Coexistence Constraint.

3.4 Correctness and Adaptability

Working with constraints in the context of KiPs means complexity and frequent changes. The former does often originate from the constraints themselves and threatens correct and desired behavior. Ways to ensure correctness when facing increased complexity and interferences are the topic of Section 3.4.1. After that, in Section 3.4.2 the discussion is on how the required adaptiveness in face of the frequently expected changes can be achieved.

3.4.1 Complexity and Interferences

When applying several constraints to a process the possibility for increased complexity grows by a significant margin. Arising problems concerning the correctness in regards to the desired function of a system need to be prevented or treated accordingly. While some of the dangers to correctness rise from the complexity coming with interplay between multiple constraints, others originate from the KiPs frequent changes.

Having a small number of constraints is usually safe as users are usually still easy to grasp the constraints' effects. Creating several dependencies for a bigger amount of work-items can increase risks substantially, though. Often, problems do arise when an entity is affected by more than one constraint - especially when constrained by one and restrained by another. Note that users can also make mistakes when setting up constraints, not correctly understanding their meaning and consequences or misinterpreting a situation for requiring a specific one to be created when it does actually not call for.

An example scenario that could lead to unwanted interferences between constraint is shown in Figure 3.33.

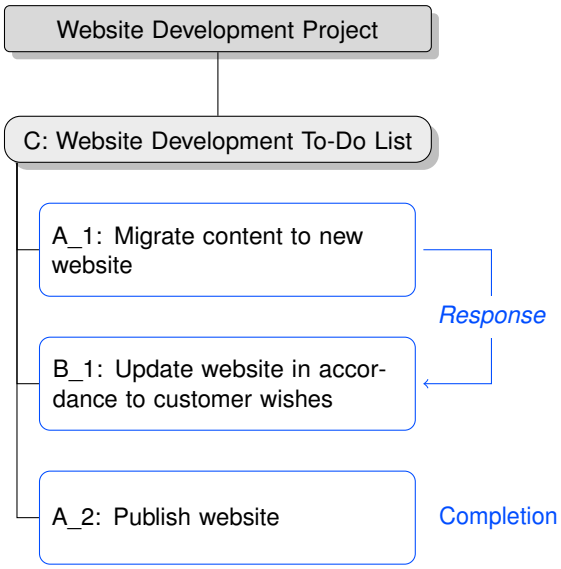


Figure 3.33: Example for undesired interferences between constraints.

Assume that a SRespCstr (see Section 3.3.2) was created for tasks A_1 and B_1 of the to-do list, restraining the context C. So after discussing the state of the website, it must be updated accordingly. Only then the to-do list may be set to *completed*. Additionally, a second ComplCstr was created for task A_2. This would mean that as soon as publishing the website is *completed*, the ComplCstr will restrain all sub-entities of C from being put into, e.g., states *running* or *completed*. The scenario get's complicated when A_1 was completed and the SRespCstr triggered, and afterwards A_2 triggered the ComplCstr. In this case, first, the to-do list is still restrained from completing by the triggered SRespCstr. Task B_1, as a sub-task of C, would be restrained from completing by the ComplCstr. With this, the SRespCstr could no longer get satisfied. Now C would be locked out from being set to *completed*.

The example describes a scenario, where combinations of existing constraints could lead to undesired interplays, with one constraint restraining state changes which would satisfy or trigger another constraint. Scenarios like this can occur for a multitude of different reasons and constraint combinations. Several solutions can be proposed in such cases. Combinations of them are thinkable, as well.

Solution 1: Correctness by Construction

The idea is to identify possible problems before a constraint is created, and notify the user to make him aware of the concerns and get a human decision. It is worth mentioning, that the problem here is partly that B_1 is for one the satisfying entity for the SRespCstr, and second a automatically detected RE for the ComplCstr due to being a sub-entity of C. This double strain on entities could be identified by querying for entities annotated for being both constrained (the former case) and restrained (the latter case) by different constraints. Still, in the example above, A_2 would be automatically detected by the ComplCstr when it sets up the restraints for sub-entities, thus rendering the approach ineffective for the example. This holds true for all cases, where automatically identified sub-entities are playing a role, as constraints only get to know these when putting restraints in place for them, and not beforehand.

Solution Check for possible problems before actually creating a new constraint. This is done by identifying SEs which have restrained states declared for one constraint and at the same time triggering and/or satisfying states declared for another constraint. Through the CEs and the REs assigned to created constraints this would be easily possible. Automatically detected sub-entities which can be affected have to be identified by new rules, though. Based on the information, users would then be able to decide if a constraint should actually be created with the chosen variables.

Required extension New rules need to identify the sub-entities of a context specified for every constraint.

Purpose of the rules Identifying the sub-entities before the constraints are created is necessary, as otherwise this information would only be gained when the restraints are actually set in place. The rule needs to store information on concerned sub-entities and keep it up-to-date.

Solution 2: Signaling Between Constraints

A second and more easily achievable solution would be through signaling between constraints. This happens after the constraints are created and is based on using signaling facts inserted by one constraint for giving message to others.

Solution Utilize facts for signaling unfinished work by one constraint so that others can take a step back for the time being if needed. The signaling fact is inserted by constraints who need to be satisfied and where needed can be checked for by constraints that could possibly prevent the satisfaction.

Required extensions A new fact type for signaling (*unfinished business fact*) and new rules (e.g., checking for this fact) need to be added.

Outline for the Unfinished Business Fact

References:

- the id of the creating constraint
- the id of the CSE that has pending work on any of it's sub-entities.

Purpose of the rules First, the Unfinished Business Fact has to be inserted and revoked. Second, other constraints have to check for it and react accordingly.

Rules for Inserting and Revoking the Unfinished Business Fact

- The moment a restraint is set up, an unfinished business fact is inserted by constraints that wait for their satisfaction. For each context that has such unfinished business pending on any of it's sub-entities a single unfinished business fact is inserted.
- The unfinished business facts get revoked together with the restraints of the constraint that set it up.

Rules Checking for Unfinished Business

- Constraints that could lead to interferences check for unfinished business facts referencing the same context they would set up restraints in. If any are found, they restrain their CEs from changing into their goal states, thus putting themselves on halt and preventing constraint state changes that could lead to the interferences. When doing so a flag is set in the CSF remarking the halt.
- When the flag that annotates the halt was set and the unfinished business fact is finally gone, the flag will be changed again, the restraints for the constraint's CEs are lifted and the constraint is thus free again to move on.

In the example above, the SRespCstr after being triggered by `A_1` has still a unsolved satisfying condition and at that moment, a restraint for `C`. The SRespCstr inserts an *unfinished business signaling fact* referencing the context (in this case `C`). Furthermore,

the ComplCstr is checking for facts of such type referencing the context it is specified to work on. Only when no such fact is found, the ComplCstr will proceed to restrain it's sub-entities. Furthermore, the ComplCstr would even put a restraints up for it's own completing entity (A_2 in the example), preventing it from going into a triggering case. When the response is satisfied and lifts it's restraints, it also removes the unfinished business signaling fact. This method comes with a drawback though, as not all types of constraints are concerned by this problem and the affected ones need to be singled out and provided with the described additions. Also, the understanding and awareness of possible interferences becomes more difficult when new types of constraints are added.

Solution 3: Constraint Solving

A much more sophisticated approach would include methods of constraint solving. Even before the creation of a constraint checks would be performed to determine if all of the (planned) constraints will be able to reach every single one of their constraint states and when this would no longer be possible. Answers could be given even before the creation of a new constraint by a user, then. Being part of the constraint satisfaction problem, this is an advanced and difficult approach, though.

3.4.2 Adaptability and Changes

Frequent changes are nature to KiPs and alterations to the structure of a process can interfere with the meaning of previously created constraints. Common cases of changes are, e.g., the addition of new work-items or moving or removing existing ones.

Another example shown in Figure 3.34 is taken into consideration.

Assume an ImmSuccCstr was created for A and B. After a website release was approved by a customer (A), the new website must then be published (B), and with no other tasks of C being started or completed in between (i.e. T). If the constraint is triggered, and a user moves task T out of the context and maybe into a different the to-do list of maybe a second team, the question arises if the restraint on T should still stay active. Also, new

Figure 3.34: Example scenario containing an ImmRespCstr.

tasks could be added to the to-do list C. These would also need to be restrained right away.

Constraints therefore have to adapt to changes at any given time. After discussing each of the problems possible cases, a couple of solutions will be presented.

Case 1: Adding New Entities

In this case the autonomy of rule evaluation solves the problem. Changes relevant to a rule like newly added tasks inserted as facts lead to the re-evaluation of a rule. A new restraint would then be put in place, automatically. The existing implementation can therefore handle the creation of new entities correctly and as expected.

Case 2: Removing Entities

When an entity is removed, there is no longer a need for any restraint on it, and existing ones are therefore removed. If the entity is the CE of some constraint, this constraint can no longer work as intended. Therefore, the constraint is removed together with

the entity. The same happens, when the last RE that was specified for a constraint is removed. Without, e.g., restraining any context until a SRespCstr is satisfied, there are no restraints on any state changes at all and thus there is no requirements to actually satisfy the constraint.

Case 3: Moving Entities

Moving entities into the context that were previously not present leads to a similar re-evaluation as in case 1. Moving entities out of the context is a bit more complex. The first problem is deciding, if the restraint is still relevant. This is pretty sure the case if the entity in question is one of the CEs, as they were chosen explicitly. Selected REs on the other hand could be wanted to have active restraints for them lifted. Automatically identified REs are always chosen due to being sub-entities of a specified context. When moved out of these context they are therefore always to be freed of restraints.

For all cases where the moved entity is explicitly specified for a constraint and not a CE, the question if restraints have to be lifted can not necessarily be answered by some form of automation. Constraints and rules are not aware of the reasoning behind the selections made, and in all such cases the user's decision is needed. In the current state, the implementation leaves the restraint remaining active and managed by the Constraint and user's have the option to manually remove unwanted restraints.

Case 4: Changes to the Variables of a Constraint

Users could decide to make changes to existing constraints that are possibly not anymore just in constraint state initialized, but have set up restraints already. Note that changes to the CEs are exlcuded from this case, as these do rather require the creation of a new constraint and possibly the deletion of the old one. For changes to any other variables, lifting no longer needed restraints can be done in two ways.

First, by removing no longer needed restraints by removing the persisted restraint objects as part of the method changing the constraint's variables. This is done on side of the traditional codes without the use of rules. All chosen variables for a constraint are

stored together with the constraint object and restraints do always reference the creating constraint's id. Thus, a connection can be easily made and the restraints be removed.

Second, when a RE being a CSE is removed from the variables of a constraint or changed, and the constraint type puts restraints on automatically identified sub-entities of this RE, restraints for these sub-entities have to be revoked. This is similar to what was stated for case 2 (see Section 3.4.2) above.

Solution 1: Automatically Remove No Longer Needed Restraints

This solution is aimed at solving problems for cases where the parent of a sub-entity that was restrained is no longer a RE to the restraining constraint. The solution is based on a new fact type and a rule identifying sub-entities concerned by the problem that will then get their restraints removed.

Solved Problems The two problems solved by this solution are listed below. Both times, the parent of a sub-entity that was restrained is no longer a RE to the restraining constraint and the restraints for the sub-entity therefore need to be revoked.

- Restrained sub-entities of a context are moved into a non-restrained context, meaning the new parenting CSE is no longer a specified RE of the constraint that set up the restraints (case 3 above, see Section 3.4.2).
- After changes to the variables of a constraint the parent of a restrained sub-entity of a context is no longer a RE to the constraint that set up the restraint (case 4 above, see Section 3.4.2).

Solution Utilize rules to detect sub-entities that need to have their restraints revoked due to changes to their parent entities.

Required extension A new fact type is needed (*restraint fact*) and two new rules for identifying the concerned sub-entities.

Outline for a New Fact Type (*Restraint Fact*)

References the following information:

- the id of the sub-entity for which one or more restraints were put in place
- the ids of the parent entities of this sub-entity at the point in time when the restraints were set up
- the restraining constraint's id

Purpose of the rule The rules added for solving the problem evaluate based on the existing facts and the new fact if the restraints for sub-entities have to be lifted.

Rules for Revoking No Longer Needed Restraints

A restraint is revoked whenever a restraint fact (see above) references only parent entities that are either all:

- no longer current parents to the restrained sub-entity (due to the moving the sub-entity) (rule 1), or
- no longer REs for the constraint that set up the restraint (rule 2).

The rules for implementing this solution could be written once globally to be shared by all types of constraints, as the case and solution are common to all types. Note that the conditions of the two rules can also be combined into only one.

Solution 2: Ask Users for New Entities to Take Over

In all cases where changes are done to entities specified as variables to some constraint but it can not be automatically decided if existing restraints for this entity should be kept in place, the question has to be forwarded to the user. Decisions like selecting new entities (see case 4, Section 3.4.2) or removing the constraint can then uphold the desired system behavior.

3.4.3 Summary

A couple of threats to correct constraint behavior and solutions to these were discussed in this section.

At first, the focus was on the increased complexity coming with the use of constraints and, based on an example, how constraints could interfere with each others work and state changes. With the consequences of having several constraints active not always being easy to grasp, signaling facts can have constraints take a step back from progressing when others need to be satisfied, first.

Next, the hurdles coming with changes to the process were laid out. The possible problems originating from cases of adding, moving and removing can be overcome by utilizing the rule engines readyness to re-evaluate when facts are changed, implementing a few additional rules and by asking the user to help in the decision making. The latter is often the most important, as a system can not always determine constraint should handle entity changes by itself.

Similar to the outcome of the analysis regarding reverting constraint states in Section 3.2.3 before, involving the users through awareness and interaction is a highly beneficial addition.

3.5 Extending with Additional Functionality

After the concept was describing the rule engine, concept for constraints and types of constraints as wells as some possible challenges and solutions through complexity and frequent changes, the topic is now on how to extend the concept with additional functionality. The features described in the following do not just give their conceptual ideas, but also provide examples of how the concept is open for fluent extension.

First, Section 3.5.1 shows how automated actions can further reap the rule engine's benefits and aid users with monitoring and representing the ongoing work. Next, constraint templates are introduced as a rule-based addition allowing users to pre-set constraints in

Section 3.5.2. And last, temporal constraints as an example of how to enrich the catalog of constraints are discussed in Section 3.5.3.

Note that the additions shown here are not part of the implementation, if not stated otherwise.

3.5.1 Automated Actions

The goal of automated actions is to aid users by automating small actions they would otherwise be required to perform themselves and that are logically expected.

With the rule engine being the driving force behind the constraints and their behavior not only is a flexible but also a very potent tool at hand that puts a strong emphasis on the easily adding further rules to it. The automated actions show how to a few simple rules based off the already existing facts and their information about the system can easily add new functionality in small, encapsulated rules.

A lot information on the system entities and their current states is already available to the rule engine through various facts used for the constraints. Without adding anything fact-wise on top of it, all that is need to realize the automated actions, is a single rule for each added automated action.

An example for such a possible actions is the automation of state changes for CSEs based on their sub-entities' states.

Example 1: Reflect Started Work on a Context

A newly instantiated task tree in the proCollab system has the task tree entity all it's sub-entities in entity state *initialized*, for example. As soon as a user changes any of the sub-entities to states *running* or *completed*, this symbolizes that the work on the task tree has begun. In order to reflect this, a user would need to change the task trees state, as well. An automated action can take over for this as shown below.

If

- any CSE is still in entity state *initialized*, and
- any of it's sub-entities is in entity state *running* or *completed*.

the rule will call the system's method for changing an entity's state and change the CSE into state *running*.

Example 2: Reflect Completed Work on a Context

Another automated action aiming at a context and it's sub-entities would change the task tree from the example above into entity state *completed* as soon as all it's sub-entities have reached the very same state.

If

- any CSE is not in entity state *completed*, and
- all of its sub-entities are in entity state *completed*

the rule will call the system's method for changing an entity's state and change the CSE into state *completed*.

The two automated actions shown here are part of the prototype. Many more automated actions could be thought of and are easy to add to the collection of rules for taking over work that is otherwise expected to be done manually by users thus helping to correctly reflect the ongoing work on the process.

3.5.2 Constraint Templates

With the major goal of providing guidance to collaborating knowledge workers, pre-defined templates are an often employed method to let other users benefit from prior work and knowledge. The proCollab system and other systems for the support of KiPs therefore make extensive use of templates for their work-items like tasks, task-trees or processes [13] [23] [25]. Creating new constraints based on constraint templates follows this close. A user could not only want to create a task-tree combining several tasks into a list, but also might want to model dependencies between some of these tasks ahead

of the tasks' instantiation. Having constraint templates at disposal helps to pass along gained knowledge to other workers who re-use the templates. The subject of this form of constraint template here is to create a template that was set for templates of entities of the system, rather than instances of such. The problem comes with the time and number of entity instantiations and it's solution can be found with rules, once more. The concept of constraint templates can therefore also be seen as an example of how to add further functionality to the existing system and the supported constraints utilizing the introduced technology.

Quite in accordance to the approach for constraints, users are provided guidance for creating from constraint templates, here. The concept for constraint templates results in offering a constraint proposal to the user. Each constraint template has a proposal object created and persisted for it that lists the constraint type and templates of entities that can be chosen as variables as soon as they have instances created for. Lists of SE instances given by the constraint proposal are used to select entities for filling the roles of the CEs and REs. More on filling this lists is shown below (Section 3.5.2).

Outline for a Constraint Template Object

References the following information:

- the constraint template's id,
- the constraint type that is to be created from it, and
- all necessary variables for this constraint type but with templates of entities picked as variables.

Outline for a Constraint Proposal Object

References the following information:

- the id of the constraint template it was created for,
- the constraint type that is to be created from it,
- all variables selected for its referenced constraint template, being templates for entities with each one referencing the entity template id, and
- for each variable a list of entity instances that were created from the specified entity template.

Whenever an entity is instantiated that has a constraint template applicable to it, the user is offered to create a constraint from it if all the required variables are supplied with fitting instantiated entities.

The Problem with Automating Constraint Creation

After the template for a constraint was created, a problem comes into view when the required instances are created one by one, instead all together like, e.g., when part of an instantiated context. As an example, consider the case where a constraint works on CE A coming from a to-do list, and CE B being part of a checklist. If both lists are instantiated together with let's say the parenting process, this will be unproblematic. Not so, when both lists are instantiated separately. In this case, we could have one instance of the to-do list and two instances of the checklist. The question would then be: Is a (second) constraint to be created between the second instantiated checklist and the first instantiated to-do list, or should there only be a (second) constraint between the second instantiated checklist and a possibly later instantiated second to-do list?

Note that in addition to the solution presented in the following, the automated creation of constraints from constraint templates in cases where all required entity templates are instantiated together, can be supported as well, of course.

The Rule-Based Solution

To solve the aforementioned problem, rules can be utilized, once again.

Solution New rules would be used for identifying the completed mappings between existing entity instances and entity templates for constraint templates. Constraint proposals then have their lists of entities for variables filled with the found entity instances.

Required extension In addition to a new rule, additional information needs to be added to the existing fact base:

- A constraint template fact for listing the required entity templates by their ids.
- Entity facts must have the id of the entity template from which they were instantiated from added to them.

Purpose of the rules Putting this information to use, a rule can then create constraint proposals for completed mappings:

- The rule has to check for the existence of a constraint template fact (`CTF`).
- If one is found, check if there is an entity fact that represents the entity instance to one of the required entity templates referenced in the `CTF`. This is then considered a successful mapping of an entity instance to an entity template.
- If a mapping was found, call a method for adding the id of the entity instance (being the id of entity fact) to the list of the constraint proposal.
- The right proposal is found by the constraint template id, referenced by both the `CTF` and the constraint proposals persisted by the system. This would be done on Java side with the method being passed the necessary parameters.

With this solution, options for creating constraints from constraint templates can be given to users: For each of the available combinations of entity instances there will be one proposal available.

On instantiating an entity, these options could be brought forth by presenting the constraint proposals referencing the id of the entity instance created, in case all required variables for this proposal have at least one instance available to choose from.

3.5.3 Temporal Constraints

In many cases dependencies bind work-items to temporal restrictions. Temporal constraints can be used to express these dependencies and are therefore a noteworthy and very useful addition to the concept. An example for adding temporal constraints in general and a specific type of those is presented in this section.

The work by Lanz et al. on the topic of time patterns ([30]) defines several types of constraints based on time patterns and was consulted for finding a couple of examples. These are only a very small selection of many more temporal constraints spread over several sub-categories and are used in this context to showcase how to extend the concept with new types.

Regarding the effect of temporal constraints, instead of only working with restraints, this concept utilizes time warning objects that present guidance in a form of reminders for delays and deadlines [26]. This is similar to the approach for temporal constraints chosen by Montali et al. for the Declare system [26]. The time warning objects would be created and persisted similar to restraints and could be either kept for later reference or removed when "consumed" (a warning was presented to a user and confirmed) or no longer needed. The system could check for these objects at various times.

Outline for a Time Warning Object

Created and persisted on side of the traditional code. Used for symbolizing warnings for a temporal constraint about to be overstepped. References at least

- Tthe creating constraint's id,
- a message notifying about the cause for the warning and the possible consequences if overstepped,
- the remaining time left,
- the ids of the entities between which the time value is placed, and
- the goal states the entities have to reach within time.

Evaluating Based on Timers and Current Time

In order to provide temporal constraints information on time we need several things: re-evaluation of rules in intervals, information on the current time and information on the points in time relevant to the temporal constraints.

For the first point, Drools offers both interval and cron based timers. They can be set per rule allowing to specify how often the rule's evaluation shall be repeated. It is also possible to define in which intervals the effect of a rule should be repeated as long as it's condition holds true. This way, temporal restraints could for example have rules available that do directly call system methods for displaying warnings in set intervals (and as long as their conditions hold true). Warnings could in some way be presented on an hourly basis this way.

For information on the current time a new *current time fact* could be used for all types of temporal constraints. Every few minutes a single rule is updating this fact that was inserted right when the Drools session was created. Changes to it would then allow the rules for temporal constraints to be evaluated to true, making use of the rule chaining mechanism.

Lastly, information for important points in time is specified as a (optional) variable for temporal constraints and has a new fact type representing it.

An example for a temporal constraint performing based on these three points is presented next.

Example: Time Lag Between Activities

With the temporal patterns for time lags it is possible to describe a time lag between two activities, either being a minimum value, a maximum value or an interval (min to max) [30]. The lags can be declared to exist between several states of the activities. The example will apply the case of such a pattern to extend one of the constraint types from the catalog (see Section 3.3) in short.

The base constraint used is the SRespCstr (see Section 3.3.2). This constraint will simply be extended with an optional *maximum time lag* between it's CEs A and B. A use case is direclty taken from [30]: "The *maximum time lag* between discharge of a patient from a hospital and sending out the discharge letter to the general practitioner of the patient should be 2 weeks". The CE A would be the discharge from the hospital with it's completion chosen as the triggering state, while CE B is sending the letter, specifying entity state completed as it's satisfying state.

Objective There is a maximum time lag between the triggering of a SRespCstr and it's satisfaction. A warning should be generated once before the maximum value is reached and once if and when it was overstepped.

Solution Utilize rules to check for the time value and timestamps after the constraint was triggered. Create time warning objects on Java side once when the first warning point was reached and once when the time value is reached or overstepped.

Required extension The SRespCstr can be used just as laid out in the concept (see Section 3.3.2) and will have an optional variable added to it, the maximum time lag value. A new fact type is needed (*maximum time lag fact*) and inserted together with the other facts for constraint variables (see Section 3.2.4) for representing this

value in the working memory. In addition, new rules are added to the existing ones for creating the time warning objects.

Outline for a New Fact Type (*Maximum Time Lag Fact*)

References the following information:

- the id of the constraint it is meant for,
- the maximum time value between the triggering and satisfaction of the constraint (the early warning point), and
- the maximum time value between the triggering and first warning (the overstep point).

Purpose of the rules The rules for creating the time warning objects compare the CSF with the current time object and the maximum time lag fact.

Rule for Creating an Early Warning

If

- CSF `CS` for the SRespCstr is found that is annotating the current constraint state as triggered and
- a time warning fact is found that references the same constraint id as `CS` and
- a current time fact is found that lists the early warning point as overstepped when compared to the current time and the timestamp for when the constraint was triggered

then the rule will call a method on the traditional code side for creating a new warning object with the according information.

The rule for creating a warning in case that the maximum time value was overstepped is working analog to the rule above, but with comparing the overstep point of the time warning fact to the other facts.

This constraint and approach is one example of how the existing concept and even constraints could be extended for new functionality.

3.5.4 Summary

Several possibilities to extend the concept presented in this work were laid out. Each follow a slightly different approach and come with different required extensions ranging from only rules to creating new facts and restraint-like objects. The examples show how new features can be added by relishing the adaptiveness of the rule-based approach and focus on facts:

- Automated actions can automate expected user actions that were otherwise needed to be done manually and help with correctly portraying the ongoing work on the process. They very likely only require a single compact new rule per automated action as all the required information is already presented through the facts for constraints.
- Constraint templates allow users to pre-define constraints for templates of system entities. Rules can help to create or fill constraint proposal objects for constraint templates as soon as all the required instantiated entities are available.
- And finally, temporal constraints show a noteworthy and completely new category of constraints. Highly valuable, these can be added by either creating new constraint types or by extending existing ones as shown.

3.6 Summary

This chapter formed the core of the work at hand:

- Rule engines in general and the Drools rule engine were introduced. Working with facts and managing the representation of the work-items was discussed, as well.
- The concept for the constraint was laid out in detail. The common representation, terms and definitions were introduced, first.

- The three core aspects of the concept were discussed next:
  - constraint states and rules,
  - facts and variables, and
  - restraints.
- After that, a catalog of selected constraints was presented.
- Correctness and adaptability were the next topic and several problems and solutions were laid out.
- Finally, several examples of how to extend the concept were elaborated on.

4

Implementation

This chapter describes the architecture and selected elements of the implementation realized as a proof of concept for the constraint's concept presented in Sections 3.2 and 3.3.

The method followed during the forging of the protoype is quickly laid out in Section 4.1. Next, the architecture is described in Section 4.2. And last, the implementation of a constraint is presented exemplarily on the case of the ImmRespCstr in Section 4.3.

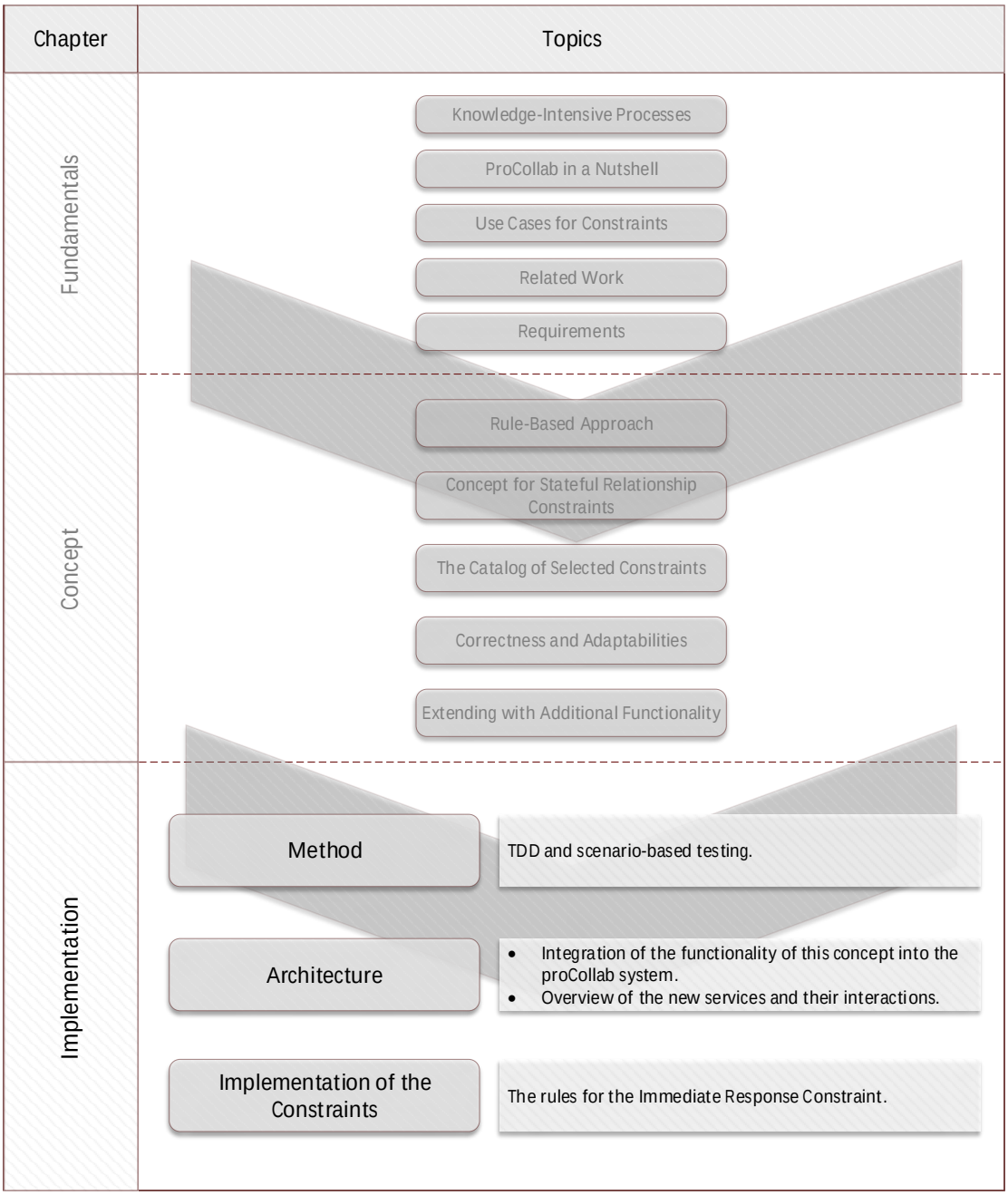


Figure 4.1: Roadmap highlighting the chapter on the implementation.

4.1 Method

The functionality for constraints and dependencies was implemented for the proCollab system following a rule-based approach utilizing the Drools rule engine. How to introduce dependencies into a system for the support of KiPs as well as further functionality to come was iteratively evaluated, implemented and refined during the agile development process.

For the implementation of the constraints the approach of test-driven development was chosen [37]. With the complexity of the logic contained within the rules, and the chaining and interaction of each of these, providing a solid base of test cases is of great help. Due to the tests written beforehand, coding the constraints' logic can be done in a controllable and laid out fashion. This was also an advantage when making changes to their new and already existing functionality.

Several new services, REST interfaces and a multitude of rules for the Drools rule engine were added to the existing proCollab system.

The method for testing was built upon a use case scenario, the strategy design pattern for validating the correctness of the work done on the scenario, and the REST Assured framework [38] for utilizing the implemented REST methods. For the scenario, the website development use case introduced in Section 2.3 was used. The strategy pattern was employed to offer a different testing strategy for each individual as well as some combined constraints. Validating that they work as intended was done by performing state changes for selected SEs of the scenario. The work on the scenario was done via calls to REST interfaces done via the REST Assured framework.

4.2 Architecture

In the following the architecture and additions to the existing proCollab system are be laid out. Beginning in Section 4.2.1, a first overview of how the new functionality was integrated into the system is given. Afterwards, Section 4.2.2 describes the newly added services for constraints and their interaction.

4.2.1 Integration into the proCollab System

An overview of the integration of the functionality for constraints into the existing system can be seen in figure 4.2. The logic for the constraints is encapsulated in rules spread out over several rule files, each file containing the rule set for one type of constraint. These rule files are part of the rule engine and not depicted in the figure.

The connection points between the existing system and the newly added functionality can be seen by the solid lines connecting the areas for each respective part. The main work for the integration therefore lies in letting users create the constraints while selecting entities and their relevant state types as variables, inserting the facts for representing the entities native to the system and checking for constraints. Further, the parts shown in the figure can be described as below.

Templates and Instances Beginning at the bottom of the figure, the templates and instances of the system entities - namely tasks, task-tress and processes - all extend the SE class.

Stateful Entities Every SE and entity inheriting from it has at least the reference state model (see Section 2.2.2) and with it multiple default states available to it. From the SEs and aforementioned sorts of instances the data relevant for the constraints proper function is extracted into entity facts which are inserted into the rule engine's working memory. The facts are created, kept up-to-date and removed by a Java interceptor reacting on methods which manage the SE. When a state change is about to be performed for a SE, the interceptor will first check for existing restraints applying to this entity and target state. For the proof-of-concept it was chosen that restraints are followed strictly. The interceptor therefore prevents restrained state changes. Note that this interceptor was already part of the existing system and hooking into it allowed to connect everything needed.

Constraints Users can create constraints choosing from the various types presented in the catalog in Section 3.3. For each new constraint a Java object is created being of a type specific to the constraint's one. Users choose their variables like, e.g., the CEs, goal states and/or restrained states by selecting from the SEs. Constraints

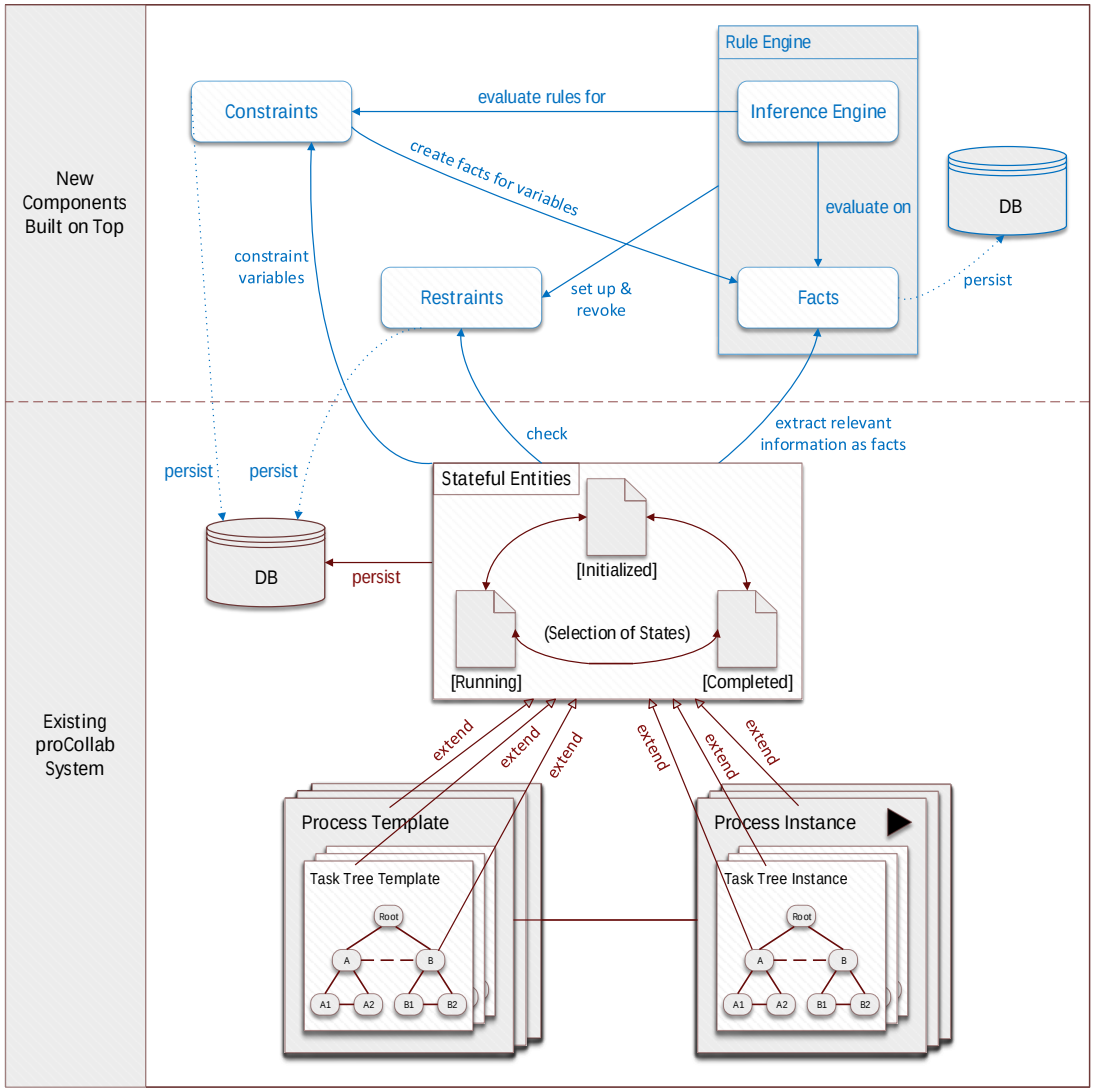


Figure 4.2: Overview of the functionality for the support of constraints and dependencies integrated into the proCollab system.

are persisted in the proCollab system's database. During the creation of a new constraint, the facts for the representation of it's variables (see Section 3.2.4) are created and inserted into the rule engine's working memory.

Facts The facts are inserted into the Drools rule engine's working memory. While herein, they and their contained information are accessible by Drools. Their whole sum, called the fact base, for one part represents the system's SE in their actual states, and for the other the variable-representing facts with their target states. Facts are persisted by Drools in a database separate from the proCollab system's one. As an alternative, it would be possible to configure Drools to use the latter one, as well.

Rule Engine (Inference Engine) The Drools rule engine is working alongside of the already existing system and evaluates the rules encapsulating the constraints' functionality by performing target-actual comparisons on the fact base. The rules' effects are responsible for setting up or revoking the restraints.

Restraints And lastly, restraints that were set up by Drools are persisted in the proCollab database. While stored there, the proCollab system checks for them whenever a SE is to change it's state (annotated by an arrow originating from the depicted SE object here). If a restraint for an entity and the state it is about to be changed into found, the implementation will strictly follow the proposal and refrain from performing the state change in question.

4.2.2 The Services and Their Roles

Several new services were created for supporting constraints in the proCollab system, as presented in Figure 4.3. Interfaces and their implementations (see above) are modeled together as single elements. The services take over different responsibilities for functionality related to constraints and each have a REST interface available for exposing it to proCollab's client. The services center around constraints, Drools and restraints and are listed below together with their purpose.

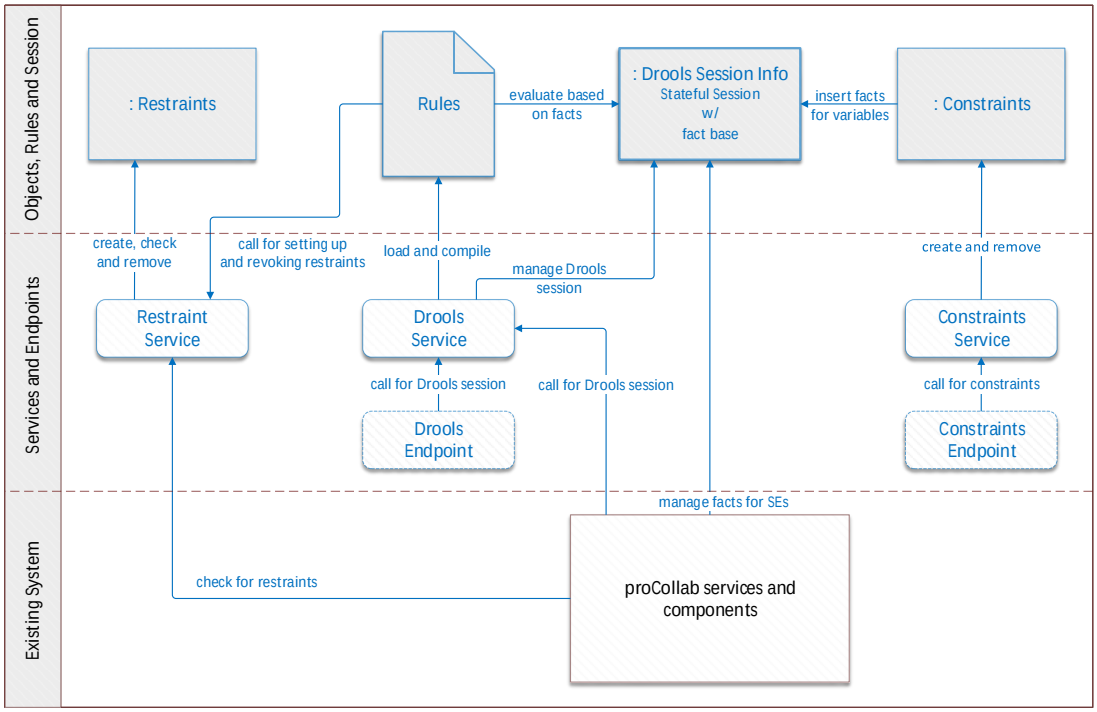


Figure 4.3: Overview of the services for constraints and their interplay.

Constraints Service Interface to Constraint Service Impl. Local Bean set up for Context Dependency Injection (CDI).

Constraints Service Impl Provides the implementation for the Constraint Service and functionality for the creation and management of constraints during their life-cycle. Stateless session bean injectable via CDI.

Constraints Service Endpoint Offers access to the functionality of the Constraint Service to clients.

Restraint Service Interface to Restraint Service Impl. Local Bean set up for CDI.

Restraint Service Impl Implements the Restraint Service's methods to set, check and revoke restraints. Stateless session bean injectable via CDI.

Drools Service Endpoint Offers access to the functionality of the Drools Service to clients.

Drools Service Interface to Drools Service Impl. Local Bean set up for CDI.

Drools Service Impl Contains the functionality to set up and manage Drools sessions as well as further Drools-related functionality implemented for the Drools Service, e.g., related to persistence. Stateless session bean injectable via CDI.

Drools Session Info A singleton storage containing data related to a Drools session that is to be held available during the execution of the proCollab prototype. The stateful knowledge session (see Section 3.1.2) stored by the singleton is a central component for Drools. The figure depicts the interaction with this session through the arrows.

Drools Agenda Event Listener A listener assigned to a running Drools session allowing to check if specific rules were fired. Mainly used for testing.

As can be seen in Figure 4.3, the existing parts of the proCollab system are responsible for creating a Drools session or loading an existing one, for checking the restraint objects and for managing the creation of facts representing native system objects. While the former two are done through the services, the latter fact management is directly done on the Drools session. This session (see Section 3.1.2) is created or loaded during the bootstrapping and referenced in the aforementioned Drools Session Info singleton - a central element used by multiple services and components. The rules are loaded from the rule files and compiled by using Drools' functionality during the session creation. Constraints know which fact types have to be inserted for the user selected variables and do so during their creation. With the session kept up and running it will then constantly evaluate rules as necessary leading to the set up and revoke of the restraints. Omitted for clarity are the calls from rules to the Constraints Service, done to annotate information on performed constraint state changes and the timestamp in the persisted constraint objects for logging purposes.

4.3 Implementation of the Constraints

Constraints are realized on two sides: the Java side with it's POJOs and the side of the Drools rule engine with multiple rules contained within a number of rule files. Each file holds the rule set necessary for the functionality of one type of constraint, as presented in the catalog of constraints in Section 3.3.

- On the Java side POJOs are used to keep knowledge of the entities a constraint should work on - either fixed, specified variables or automatically identified sub-entities of a context. The POJOs also provide functionality to create the fact types representing the different variables (like, e.g., CEs) necessary to breath life into the constraints and rules.
- On the side of the rule engine, a number of rules encapsulate the logic describing the constraints effects and conditions for when these effects should be triggered.

Section 4.3.1 will tend to the removal of constraints, first, followed by the example of a constraint's implementation for the case of a ImmRespCstr in Section 4.3.2.

4.3.1 On Removing Constraints

Constraints can be removed at any time, no matter if they were triggered, satisfied or haven't done anything at all, yet. For this, three things are done: the facts for the constraint are revoked from the working memory, restraints that were persisted for the constraint get deleted and the Java object representing the constraint itself is removed.

To retract any facts that might have been previously inserted, one or more *removal rules* shared across all constraints are invoked. These rules, stored in a separate rule file, get evaluated when a removal signaling fact referencing the constraint's id is inserted into the working memory. The insert is done on the Java side at the beginning of the removal procedure, for which the Constraints Service (see Section 4.2.2) is holding the necessary methods. Next, any restraints put in place by the constraint will be removed, so that none remain lingering. The signaling fact is then removed again, which happens

on Java side, as well. And lastly, the persisted Java Object representing the Constraint will be deleted.

4.3.2 The Example of the Immediate Response Constraint's Implementation

To close in on the details of the constraints' implementation, the ImmRespCstr (see Section 3.3.2 will see a closer inspection below. It provides a good example as it is one of the constraints coming with the broadest impact. It works on two CE A and B and does not only restrain one or more given CSEs but also all sub-entities within these contexts. Figure 4.4, repeated from Section 3.3.2 earlier, is describing the concept for the ImmRespCstr.

The different rules making up the ImmRespCstr are stored in a rule file separate from those for other constraint types. The rules for this constraint are summed up in Figure 4.5 and will be discussed below. Note that they are abbreviated for clarity and have, e.g., the parts for logging constraint state changes omitted. Each of the rule shows one step towards setting up and revoking the restraints needed to express the dependency between the two CEs.

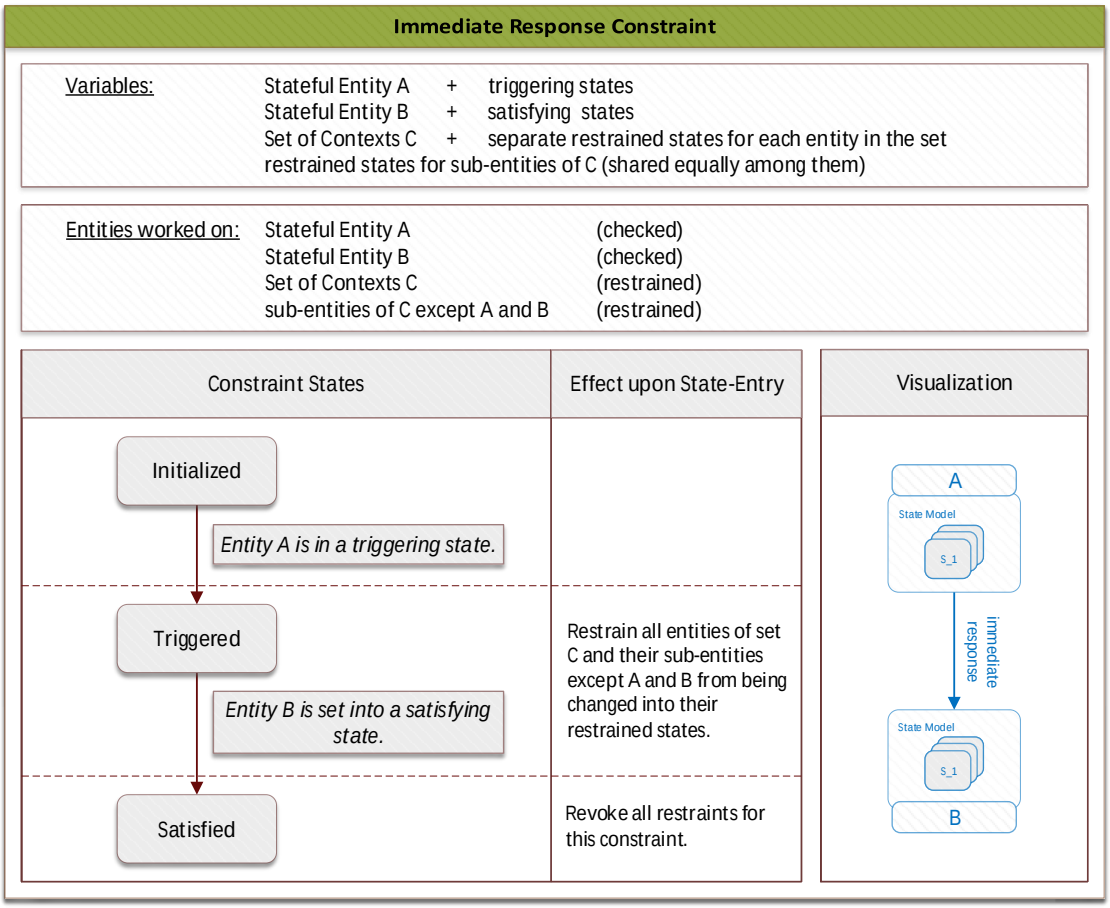


Figure 4.4: Concept for the Immediate Response Constraint (repeated).

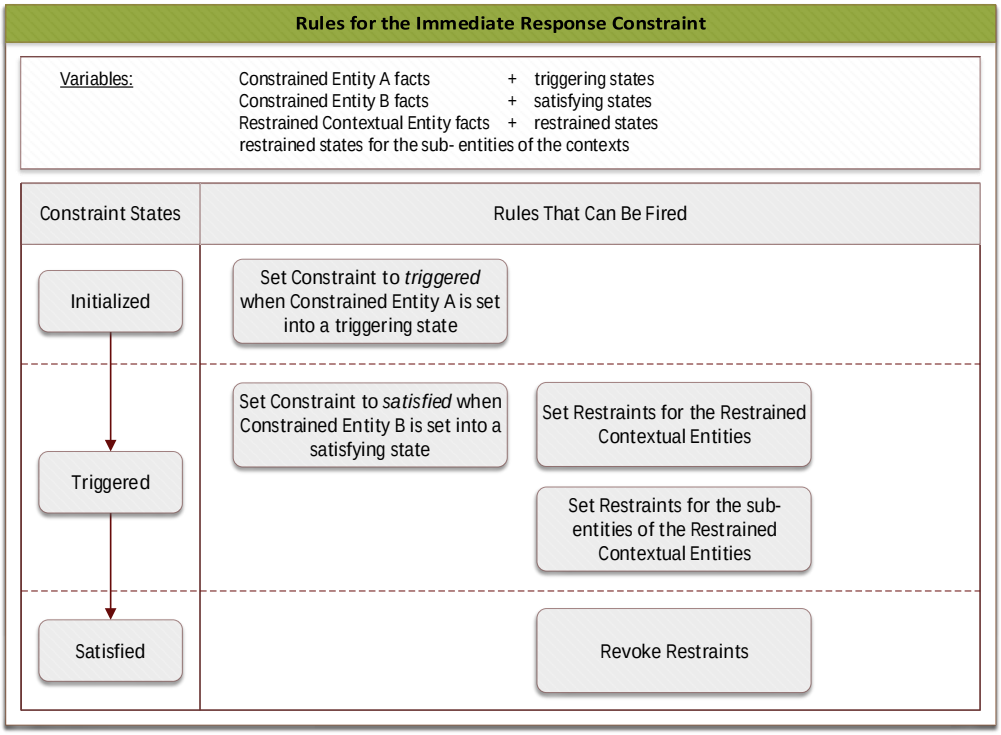


Figure 4.5: Rules for the Immediate Response Constraint.

Rules for Changing the Constraint State

The *constraint states* are used for two things: the semantic description of the constraints progress and as a mechanism to allow further rules to fire via rule chaining. Rules for changing the ImmRespCstr's state are shown below.

Initialized to Triggered The following rule (Listing 4.1) decides when to put an initialized ImmRespCstr into constrained state *triggered*.

Line 3 A Response Constraint State fact is referencing the constraint it represents by it's id and holds it's current constraint state. This state has to currently be set to *initialized* for the rule's condition to be evaluated to true.

Line 4 The rule will look for any Constrained Entity A facts linked to the constraint by the constraint's id. These facts specify the ids of CEs of set A. Additionally, they

hold one target entity state per fact (here the entity state selected as triggering states for the constraint).

Line 5 If an entity fact is found with the same id as the Constrained Entity fact's id, the former has to be the representation of the actual CE A as currently in the system. This entity fact holds the actual state. A comparison is now done between the target entity state and the actual entity state.

Line 7 In case the condition on the LHS is true (at least one Constrained Entity A is in a triggering state), the Constraint State fact is modified so it reflects the constraint's state as changed to *triggered* together with an updated time stamp.

```
1 rule "response_immediate_triggered"
2     when
3         $CS : RespImmCstrStateFact ( constraintState == ConstraintState.INITIALIZED )
4         $CE : ConstrainedEntityAFact ( constraintId == $CS.constraintId )
5         $EF : EntityFact ( id == $CE.entityId, currentState == $CE.triggeringState )
6     then
7         modify ( $CS ){ setConstraintState( ConstraintState.TRIGGERED ), setDateUpdated
              ( $EF.getDateUpdated() ) };
8 end
```

Code Snippet 4.1: Rule moving an Immediate Response Constraint from *initialized* to *triggered*.

Triggered to Satisfied The rule below (Listing 4.2) decides when to put a *triggered* ImmRespCstr into constraint state *satisfied*.

Line 3 A Response Constraint State fact is referencing the constraint it represents by it's id and holds it's current constraint state. This state has to currently be set to *satisfied* for the rule's condition to be evaluated to true.

Line 4 The rule will look for any Constrained Entity B facts linked to the constraint by the constraint's id. These facts specify the ids of CE B. Additionally, they hold one target entity state per fact (here the entity state selected as satisfying states for the constraint).

Line 5 If an entity fact is found with the same id as a Constrained Entity fact's id, the former has to be the representation of the actual CE B as currently in the system.

This entity fact holds the actual entity state. A comparison is now done between the target entity state and the actual entity state.

Line 7 In case the condition on the LHS is true (at least one CE B is in a satisfying state), the Constraint State fact is modified so it reflects the constraint's state as changed to *satisfied* together with an updated time stamp.

```
1 rule "response_immediate_satisfied"
2     when
3         $CS : RespImmCstrStateFact ( constraintState == ConstraintState.TRIGGERED )
4         $CE : ConstrainedEntityBFact ( constraintId == $CS.constraintId )
5         $EF : EntityFact ( id == $CE.entityId, currentState == $CE.satisfyingState,
              dateUpdated > $CS.dateUpdated )
6     then
7         modify ( $CS ){ setConstraintState( ConstraintState.SATISFIED ), setDateUpdated
              ( $EF.getDateUpdated() ) };
8 end
```

Code Snippet 4.2: Rule moving an Immediate Response Constraint from *triggered* to *satisfied*.

Rules Firing in Certain Constraint States

With the states of the constraint used as part of the rule chaining mechanism, the following rules handle the setting and revoking of restraints depending each on a certain constraint state.

Restraining State Changes for the Context With the ImmRespCstr signaled to be *triggered*, this next rule (Listing 4.3) puts up the restraints for the restrained CSEs. Similar to the CEs, the CSEs for which this constraint should put up restraints can be selected by the user and are inserted as Restrained Contextual Entity facts (a subclass of Restrained Entity). This rule will fire once for each fitting Restrained Entity fact found and put the required restraint in place.

Line 3 A Constraint State fact is referencing the constraint it represents by it's id and holds it's current constraint state. This state has to currently be set to *triggered* for the rule's condition to be evaluated to true.

Line 5 The rule will look for any Restrained Contextual Entity facts linked to the constraint by the constraint's id. These facts specify the ids of the REs. Additionally, they hold one restrained state per fact.

Line 6 In case the condition on the LHS is true, an external function of the Restraints Service is called to put the restraint for the specified SE and it's restrained state in place.

```
1 rule "response_immediate_restrain_context"
2     when
3         $CS : RespImmCstrStateFact ( constraintState == ConstraintState.TRIGGERED )
4         $REF : RestrainedContextualEntityFact ( constraintId == $CS.constraintId )
5     then
6         restraintService.setUpRestraint( $REF.getEntityId(), $CS.getConstraintId(),
              $REF.getRestrainedState() );
7 end
```

Code Snippet 4.3: Rule setting up the restraints for contextual stateful entities chosen for an Immediate Response Constraint.

Restraining State Changes for the Contexts' Child Entities Further restraints are set up for the sub-entities of the specified CSEs. This task is handled by the rule below (Listing 4.4). The affected sub-entities are not selected by a user and chosen by the rule based on parent-child relations between entities in the system and the user-selected restrained CSEs. The restrained states are shared between all of the sub-entities. This rule will fire once for each fitting sub-entity found and put the required restraint in place. Note that matching on a number of facts and fields as high as in this case is rather the exception than the rule.

Noteworthy here is the new fact type, the *Context Relation Fact*, shown below. It serves a special purpose as it provides information on the relation between a SE and a CSE without the fact being laden with unnecessary information. On Java side, the SEs each hold a collection with references to all their corresponding parent entities whereas for Drools more compact facts are used. The Context Relation Fact holds the following information:

- the id of the child entity and
- the id of the parent entity to the child.

The maintenance for these facts is part of the corresponding SE's maintenance. The entity listener used not only inserts, updates and deletes entity facts for SEs, but also manages the Context Relation Facts for those.

The rule fires once for each identified child entity and restrained state.

Line 3 A Constraint State fact is referencing the constraint it represents by it's id and holds it's current constraint state. This state has to currently be set to *triggered* for the rule's condition to be evaluated to true.

Line 4 The rule will look for any Restrained Contextual Entity facts linked to the constraint by the constraint's id. These facts specify the ids of the restrained contextual entities.

Lines 5 and 6 The CEs A and B are identified.

Line 7 All Context Relation Facts are identified that reference one of the Restrained Contextual Entities as a parent. In addition, they must be different from Constrained Entities A and B.

Line 8 The restrained states shared for all the child entities identified before are provided through another fact (Restrained Child State Fact). As this fact holds only the restrained states for the sub-entities for this constraint, this is just one way to find the states to disallow. Another valid way in this case where states are used equally for all sub-entities would be to set the states as fields of the constraint's POJO. The function called by the RHS of the rule could then check for the constraint object and the restrained states on Java side.

Line 11 In case the condition on the LHS is true, an external function of the Restraints Service is called to put the restraint for the specified state of the child entity in place.

Lifting Restraints When the ImmRespCstr is in constraint state *satisfied*, any restraints possibly put in place before are lifted by the rule shown in Listing 4.5. Which exact restraints have to be lifted (by deleting the persisted objects for them) is determined

```
1  rule "response_immediate_restrain_subentities_of_context"
2      when
3          $CS : RespImmCstrStateFact ( constraintState == ConstraintState.TRIGGERED )
4          $RCE : RestrainedContextualEntityFact ( constraintId == $CS.constraintId )
5          $CE_A : ConstrainedEntityAFact ( constraintId == $CS.constraintId )
6          $CE_B : ConstrainedEntityBFact ( constraintId == $CS.constraintId )
7          $CR : ContextRelationFact ( childEntityId (!= $CE_A.entityId && != $CE_B.
               entityId), contextId == $RCE.entityId )
8          $RCS : RestrainedChildStateFact ( constraintId == $CS.constraintId )
9      then
10         restraintService.setUpRestraint( $CR.getChildEntityId(), $CS.getConstraintId(),
               $RCS.getRestrainedState() );
11 end
```

Code Snippet 4.4: Rule setting up the restraints for sub-entities of the contextual stateful entities chosen for an Immediate Response Constraint.

by the Restraint Service. As restraints reference the constraint that brought them into existence by it's id, only the constraint's id has to be passed to the service. The Restraint Service then lifts all restraints referencing the constraint id.

Line 4 A Constraint State fact is referencing the constraint it represents by it's id and holds it's current constraint state. This state has to be set to *satisfied* for the rule's condition to be evaluated to true.

Line 7 In case the condition on the LHS is true, an external function of the Restraints Service is called to revoke any restraints that were put in place by the referenced constraint before.

```
1  rule "response_immediate_reallow"
2      when
3          $CS : RespImmCstrStateFact ( constraintState ( == ConstraintState.SATISFIED ||
               == ConstraintState.INITIALIZED ) )
4      then
5          restraintService.removeRestraintsForConstraint( $CS.getConstraintId() );
6  end
```

Code Snippet 4.5: Rule for lifting all restraints of a Branched Response Constraint.

4.4 Summary

In this chapter the implemented services and components that make up the prototype for the constraint's concept were laid out. It was shown how they are built on top of the

proCollab system to extend with the constraint related functionality. Through this, the prototype supports all constraints shown in the previous catalog of Section 3.3.

An example for the implementation of the rules were given for the ImmRespCstr, setting up restraints for one or more contexts and their sub-entities while utilizing refined fact types.

Despite that the information shown in this chapter grants just a quick glimpse on what was done for the proCollab system, it can be taken as an example and idea of how to adapt the concept for an information system, and for how to structure a constraint's rules and employ the various fact types for variables and system entities.

5

Summary and Outlook

5.1 Summary

Knowledge-intensive processes are a dominant sight in a vast variety of fields today, ranging from the automobile industry to healthcare and law, software development and many more. Expertise and experience of knowledge workers are earned over time and knowledge is continuously added to a process while work on it is performed. These acquisitions have to be shared with others, only then the combined contributions towards a common goal can lead to a successful outcome. Supporting the individuals partaking in the modern world's valued knowledge work through adequate software solutions designed with the specific purpose of meeting the challenges of their uncertain but sure to grow and frequently changing tasks is high on the agenda. Yet, sufficiently well performing and practical information systems are scarcely found. Many are still being developed and improved on through ongoing efforts and research. An important aspect of the knowledge at stake are the dependencies spanning across the ever-growing amount of work-items. The work at hand is doing it's part and adds the share of delivering a concept for the support of constraints and dependencies in knowledge-intensive processes to the landscape of systems.

The inspection of related work in Chapter 2 helped to find the fundamental challenges found when working on the process type in question and what features and qualities suitable supporting systems are built upon to face these. The latter is important to this work as it lied out how to built a concept applicable to many different software solutions. The scenario of a website development projects was presented and the use cases for constraints found within were met throughout the remaining chapters. Related and

influential work was discussed next that helped to base this concept on profound ground established through extensive research and finally, the requirements for the concept at hand were listed.

Chapter 3 formed the main part detailing how the concept is designed beginning on the Drools rule engine as the driving force for the required logic and functionality. The use of this technology allows for a naturally fitting, rule-based and declarative approach, leading to the formulation of compact rules and hands-off adaptability. Succeedingly and based on this, the concept for the support of constraints was presented, detailing it's state-centric method, how facts allow for adaptability and the meaningful aspect of providing guidance to the users and the system. This guidance can be considered the main focus of this concept, as instead of automation and strict ruling the final choice deemed necessary for practical applications has to be placed on side of the users. Next, a catalog of constraints is given that forms a solid and broad base for establishing dependencies. They come comprehensible and with clear semantic meanings and representations while explicitly tailored towards being used with contextual organized, stateful and task-centric work items. To make sure the concept can fully meet the required readiness for working on frequently changing processes, several challenges and solutions regarding correctness and adaptability were addressed in the following. Rounding out this chapter, several examples like automated actions, template support for constraints and new constraint types showed how to easily extend the concept and also the constraints presented within for expanded functionality.

As a proof-of-concept a prototype was developed, implementing the constraints as presented for the proCollab system. Several insights into the services written for it and the approach of how to integrate this concept into an existing system are handed out in Chapter 2. On the example of a constraint it could be seen how the rules for this logic-heavy concept are implemented, effective but still seizable, following a clear and comprehensive pattern.

Summing up, this concept delivers a solution for supporting constraints in KiPs that puts it's main focus on three points: practical usability, readiness for extension and adaption, and a strong emphasis on guidance over strict ruling. By applying the approach and

work laid out within here, many systems can have constraints working alongside of their existing functionality, expressing relationships between task-centric work-items, aiding users and knowledge workers to contribute and coordinate based upon them. Despite the depth and width of the information presented here, only parts of the implementation and only some of the thoughts on the topic could be discussed here. Much more could be added and such the work and research stays open for future enhancement and improvement.

5.2 Outlook

With the work done for this concept the proCollab system was extended with added functionality for the support of an already extensive set of different constraints ready to see use in it's processes and task lists as well as capabilities to automate smaller context related actions through the integration of the rule engine. Still, the concept focuses on the back-end, exclusively and much more can be done to increase it's readiness for similar systems and KiPs even further.

A first and especially valuable step would be to include the user in the decision-making and steering of the systems functionality, resulting in tremendously helpful fine-tuning of hard or impossible to automate behavior. This not only aids in cases in which the man-made constraints see changes to the process they have been declared for but even more so amplifies the much important user awareness. Knowledge workers are expected to benefit greatly from having existing constraints and their consequences and meanings visualized so they can better steer their work and reassess their actions.

With the rule engine a might tool with vast capabilities sits at the heart of this concept. It can be extended with far more functionality. This means more automated actions that even when being small can help with correctly representing and monitoring the ongoing work, reducing the need for manual interactions and streamlining what the system can display and offer.

Further useful extensions are putting the constraints' creation and management under the oversight of (role based) access control and making constraints mandatory or optional

(similar to [25]) by adding a new variable. By this, systems have multiple factors they can take into consideration when deciding how strictly or loosely the proposals resulting from a constraint's consequence or effect should be followed and what more can be done and created.

Extensive testing for correctness and performance would lead to more insight on how the concept could be fine-tuned for different scenarios while avoiding undesired results originating from the ever changing processes and increased complexity when having a multitude of constraints created for their tasks.

And finally, the catalog of constraints can be extended with new categories and types to cover even more uses cases.

Future research on KiPs, on systems supporting such and on features like the constraints will improve the concept and related subjects for practical use in our modern, knowledge-driven world.

List of Figures

List of Tables

List of Code Snippets

Glossary and Acronyms

Branched Coexistence Constraint (BrCoexCstr) For the BrCoexCstr there is a dependency between two sets of stateful entities SES_A and SES_B requiring that when any SE of one of the sets is put into a triggering state (triggering the constraint), any SE from the other set must be put into a satisfying state at any time afterwards (satisfying the constraint). When the constraint is triggered, the entities of a set of one or more specified CSEs CSES_C will be restrained from being changed into their restrained states, until the constraint is satisfied.

Branched Precedence Constraint (BrPrecCstr) For the BrPrecCstr there is a dependency between two sets of stateful entities SES_A and SES_B requiring that for any SE from SES_B to be set into a restrained state, any SE from SES_A must have been set in a goal state at any time before.

Branched Response Constraint (BrRespCstr) For the BrRespCstr there is a dependency between two sets of stateful entities SES_A and SES_B requiring that when any SE from SES_A is set into a triggering state (triggering the constraint), any SE from SES_B must be set into a satisfying state at any time afterwards (satisfying the constraint). When the constraint is triggered, the entities of a set of one or more specified CSEs CSES_C will be restrained from being changed into their restrained states, until the constraint is satisfied.

Branched Succession Constraint (BrSuccCstr) For the BrSuccCstr there is a dependency between two sets of stateful entities SES_A and SES_B requiring that for any SE of SES_B to be set into a restrained state, any SE of SES_A must have been set into a goal state at any time before (this is the precedence part of the constraint). In addition, there is a dependency between SES_A and SES_B requiring that when any SE of SES_A is set into a goal state, any SE of SES_B must be set into a satisfying state at any time afterwards (this is the response part of the constraint). When the response part is triggered, the entities of a set of one or more specified CSEs CSES_C will be restrained from being changed into their restrained states, until the constraint is satisfied.

Completion Constraint (ComplCstr) For the ComplCstr a specified SE A has to be set into a certain satisfying state before a set of one or more specified CSEs CSES_C can be changed into their restrained states. Furthermore, after A was set into a satisfying state, none of the automatically identified stateful sub-entities of CSES_C may be set into a restrained state, anymore.

Constrained Entity An entity with a dependency expressed through a constraint. Selected by a user as either the A or B entity for a constraint and being monitored and/or restrained by it. Has triggering, satisfying and/or restrained states specified, selected from its state model..

Constrained Entity Fact Fact type representing the variable for a constraint, namely a CE. References the constraint id, the id of a stateful entity (SE) and one triggering state, satisfying state or restrained state for the referenced SE. Refined sub-types of this fact are available for differentiating between CEs A and B.

Constraint Within the context of this work, a constraint describes the dependencies between stateful, task-centric work-items and provides guidance for performing work on them by proposing restraints on changing their states and, where needed, the states of their context(s) and their sub-items. The proposed restraints are set up and revoked based on the states of work-items connected through various dependencies or relationships.

Constraint state Used for signaling which rules are allowed to fire. When *triggered* or *satisfied*, depending on it's type, a constraint can set up and/or revoke restraints. The standard progression through the available constraint states goes from *initialized* into *triggered* and then into *satisfied*.

Constraint State Fact Fact type representing a constraint's current state. References the constraint id, the constraint's current constraint state and the timestamp of the last constraint state change. Refined sub-types of this fact are available for differentiating between specific constraint types. Simple and branched versions of the same constraint type share their refined CSF type.

Context Area of application for constraints and dependencies formed by a CSE. Covers the CSE and all of its stateful sub-entities.

Context Dependency Injection (CDI) "Contexts and Dependency Injection (CDI) enables your objects to have their dependencies provided to them automatically, instead of creating them or receiving them as parameters. CDI also manages the lifecycle of those dependencies for you." [39].

Contextual stateful entity Stateful entity acting as the parent entity of a tree or list like structure and holding related stateful sub-entities. Sub-entities can again be contextual stateful entities.

Critical Path Method (CPM) Project Management tool used for planning, scheduling and monitoring project progress. Identifies the sequence of a project's activities that when delayed would also delay the completion of the project itself.

Dependency a relationship between work-items such that one cannot reach a certain state until one or more other work-items have reached specific states..

Drools Business rule management system with a forward-chaining and backward-chaining inference based rules engine.

entity fact A fact type representing an instance of a proCollab system object in it's current state.

Fact Representation of an object made accessible to the rule engine so that it can evaluate on. All facts combined form the fact base. The Drools rule engine stores it's facts in the working memory.

Goal state A summarizing term used for triggering states and satisfying states as well.

Immediate Response Constraint (ImmRespCstr) For the ImmRespCstr there is a dependency between two stateful entities SE_A and SE_B requiring that when SE_A is set into a triggering state (triggering the constraint), SE_B must be set into a satisfying state immediately afterwards (satisfying the constraint). When the constraint is triggered, the entities of a set of one or more specified CSEs CSES_C and all their automatically identified stateful sub-entities will be restrained from being changed into their restrained states, until the constraint is satisfied.

Immediate Succession Constraint (ImmSuccCstr) For the ImmSuccCstr there is a dependency between two stateful entities SE_A and SE_B requiring that for SE_B to be set into a restrained state, SE_A must have been set into a goal state immediately before (this is the precedence part of the constraint). In addition, there is a dependency between SE_A and SE_B requiring that when SE_A is set into a goal state, SE_B must be set into a satisfying state immediately afterwards (this is the response part of the constraint). When the response part is triggered, the entities of a set of one or more specified CSEs CSES_C and all their automatically identified stateful sub-entities will be restrained from being changed into their restrained states, until the constraint is satisfied.

Init Constraint (InitCstr) For the InitCstr a specified SE has to be set into a certain satisfying state before a set of one or more specified CSEs and their automatically identified stateful sub-entities can be changed into their restrained states.

Knowledge "[...] is a fluid mix of framed experiences, values, contextual information, and expert insights that provides a framework for evaluating and incorporating new experiences and information. It originates and is applied in the minds of knowers. In organizations, it often becomes embedded, not only in documents or repositories, but also in organizational routines, processes, practices, and norms." [8].

Knowledge work (KW) "[...] is comprised of objectifying intellectual activities, addressing novel and complex processes and (work) results, which require external means of control and a dual field of action." [9].

Knowledge workers "[...] have high degrees of expertise, education, or experience, and the primary purpose of their jobs involves the process and accomplishment of knowledge work." [2].

Knowledge-intensive Business Processes (KiBPs) “Knowledge-intensive processes (KiBPs) are processes whose conduct and execution are heavily dependent on knowledge workers performing various interconnected knowledge intensive deci-

sion making tasks. KiBPs are genuinely knowledge, information and data centric and require substantial flexibility at design- and run-time." [4].

Knowledge-intensive Processes (KiPs) See Knowledge-intensive Business Processes (KiBPs).

Left hand side (LHS) The left hand side of a rule containing it's condition. Specified by Drools' keyword "when".

Negated Branched Coexistence Constraint (NegBrCoexCstr) For the NegBrCoexCstr there is a dependency between two sets of stateful entities SES_A and SES_B requiring that when any SE of one of the sets is put into a triggering state, none of the other set's SEs must be put into a restrained state at any time afterwards.

Negated Branched Precedence Constraint (NegBrPrecCstr) See NegBrSuccCstr.

Negated Branched Response Constraint (NegBrRespCstr) See NegSSuccCstr.

Negated Branched Succession Constraint (NegBrSuccCstr) For the NegBrSuccCstr there is a dependency between two sets of SEs SES_A and SES_B requiring that when any SE of SES_B is to be set into a certain restrained state, no SE of SES_A must not have been set into a specific triggering state at any time before. Reversely, when any SE of SES_A has been set into a triggering state, no SE of SES_B can be set into a restrained state at any time after. The NegBrPrecCstr, NegBrRespCstr and NegBrSuccCstr are effectually equal to each other.

Negated Coexistence Constraint (NegSCoexCstr) For the NegSCoexCstr there is a dependency between two stateful entities SE_A and SE_B requiring that when any of them is set into a triggering state, the other must not be set into a satisfying state at any time afterwards.

Negated Precedence Constraint (NegSPrecCstr) See NegSSuccCstr.

Negated Simple Response Constraint (NegSRespCstr) See NegSSuccCstr.

Negated Simple Succession Constraint (NegSSuccCstr) For the NegSSuccCstr there is a dependency between two SEs SE_A and SE_B requiring that when SE_B

is to be set into a certain restrained state, SE_A must not have been set into a specific triggering state at any time before. Reversely, when SE_A has been set into a triggering state, SE_B can not be set into a restrained state at any time after. The NegSPrecCstr, NegSRespCstr and NegBrSuccCstr are effectually equal to each other.

Pattern matching Checking of specific sequences of symbols or "tokens" for the presence of a certain pattern. The match has to be exact.

Precedence Diagramming Method (PDM) Project Management method used for sequencing a project's activities. Provides a way to schedule the project and to track and communicate it's status. Offers more types of constraints than the CPM.

process-aware Support for Collaborative Knowledge Workers (proCollab) "The proCollab (process-aware Support for Collaborative Knowledge Workers) approach was developed to establish a generic, lightweight and lifecycle-based task management support for KiPs. In this context, the approach is capable to provide customizable coordination support for a wide range of KiPs in the shape of projects, cases or investigations. Further, all kind of prospective (to-do lists) and retrospective task lists (checklists) are supported to enable better task coordination and synchronization within the knowledge-intensive processes. [...] As a foundation, proCollab employs processes, task trees and tasks as its key components. To establish the required lifecycle-based support, knowledge workers may define process, task tree, and task templates in proCollab. These templates enable the definition of best practice for planning and coordination as well as the preservation of existing process knowledge. At run time, knowledge workers may instantiate specified templates or create process, task tree, and task instances from scratch." [12].

proCollab system object This collective term sums up the following work-items native to the proCollab system: process instance, process template, task tree instance, task tree template, task instance and task template.

Restrained Entity An entity either chosen by a user or automatically picked by a rule for being restrained by a constraint. Has restrained states specified, selected from it's state model.

Restrained Entity Fact Fact type representing the variable for a constraint, namely a RE. References the constraint id, the id of a stateful entity (SE) and one restrained state for the referenced SE. Refined sub-types of this fact are available for differentiating between, e.g., contextual and non-contextual restrained entities.

Restrained state A state specified for a CE or a RE that specifies a state that this entity should not be changed into.

Restraint Restraints offer guidance to a system and it's users by proposing state changes for entities to refrain from. To represent restraints, Java objects are created and persisted on Java side as called for by rules. Each restraint refers the id of the creating constraint, the entity to be restrained by it's id and one of the states this entity is restrained from being changed into. Restraints can be checked for by the system which can then decide how to act on it first.

Right hand side (RHS) The right hand side of a rule containing it's effect. Specified by Drools' keyword "then".

Rule chaining A rule's effect leading to the successful evaluation of another rule's condition. Thus the latter is then scheduled for firing. Rule chaining is a common mechanism of rule engines. Through it, a chain of rules ready to fire in sequence is established.

Rule engine "Methods and apparatus, including computer program products, for inference processing in a fact-based business automation system, including receiving a rule set as a single package, generating a dependency graph for the rule set, and generating a sequence of processing logic for optimal processing of inputted facts" [31].

Satisfying state A state specified for a CE that leads to the constraint being satisfied (moving into constraint state "satisfied") upon this entity being changed into it.

Simple Coexistence Constraint (SCoexCstr) For the SCoexCstr there is a dependency between two stateful entities SE_A and SE_B requiring that when any of them is set into a triggering state (triggering the constraint), the other must be set into a satisfying state at any time afterwards (satisfying the constraint). When the constraint is triggered, the entities of a set of one or more specified CSEs CSES_C will be restrained from being changed into their restrained states, until the constraint is satisfied.

Simple Precedence Constraint (SPrecCstr) For the SPrecCstr there is a dependency between two stateful entities SE_A and SE_B requiring that for SE_B to be set into a restrained state, SE_A must have been set into a satisfying state at any time before.

Simple Response Constraint (SRespCstr) For the SRespCstr there is a dependency between two stateful entities SE_A and SE_B requiring that when SE_A is set into a triggering state (triggering the constraint), SE_B must be set into a satisfying state at any time afterwards (satisfying the constraint). When the constraint is triggered, the entities of a set of one or more specified CSEs CSES_C will be restrained from being changed into their restrained states, until the constraint is satisfied.

Simple Succession Constraint (SSuccCstr) For the SSuccCstr there is a dependency between two stateful entities SE_A and SE_B requiring that for SE_B to be set into a restrained state, SE_A must have been set into a goal state at any time before (this is the precedence part of the constraint). In addition, there is a dependency between SE_A and SE_B requiring that when SE_A is set into a goal state, SE_B must be set into a satisfying state at any time afterwards (this is the response part of the constraint). When the response part is triggered, the entities of a set of one or more specified CSEs CSES_C will be restrained from being changed into their restrained states, until the constraint is satisfied.

Stateful entity Task-centric entity with some kind of state model and a current state available to it.

Triggering state A state specified for a CE that leads to the constraint being triggered (moving into constraint state "triggered") upon this entity being changed into it.

Working memory Part of a rule engine and storage for its facts.

Bibliography

[1] N. Mundbrod, F. Beuter, and M. Reichert, “Supporting knowledge-intensive processes through integrated task lifecycle support”, in *2015 IEEE 19th International Enterprise Distributed Object Computing Conference*, 2015, pp. 19–28, DOI: 10.1109/EDOC.2015.13.

[2] N. Mundbrod, J. Kolb, and M. Reichert, “Towards a system support of collaborative knowledge work”, in *1st Int’l Workshop on Adaptive Case Management (ACM’12), BPM’12 Workshops*, ser. LNBIP, Springer, 2012, pp. 31–42, [Online]. Available: http://dbis.eprints.uni-ulm.de/837/.

[3] N. Mundbrod and M. Reichert, “Process-aware task management support for knowledge-intensive business processes: Findings, challenges, requirements”, in *2014 IEEE 18th International Enterprise Distributed Object Computing Conference Workshops and Demonstrations*, 2014, pp. 116–125, DOI: 10.1109/EDOCW.2014.26.

[4] R. Vaculin, R. Hull, T. Heath, C. Cochran, A. Nigam, and P. Sukaviriya, “Declarative business artifact centric modeling of decision and knowledge intensive business processes”, in *2011 IEEE 15th International Enterprise Distributed Object Computing Conference*, 2011, pp. 151–160, DOI: 10.1109/EDOC.2011.36.

[5] C. Di Ciccio, A. Marrella, and A. Russo, “Knowledge-intensive processes: Characteristics, requirements and analysis of contemporary approaches”, *Journal on Data Semantics*, vol. 4, no. 1, pp. 29–57, 2015, ISSN: 1861-2040, DOI: 10.1007/s13740-014-0038-4.

[6] R. Lenz and M. Reichert, “It support for healthcare processes – premises, challenges, perspectives”, *Data & Knowledge Engineering*, vol. 61, no. 1, pp. 39 –58, 2007, Business Process Management, ISSN: 0169-023X, DOI: https://doi.org/10.1016/j.datak.2006.04.007.

[7] Öykü Işik, W. Mertens, and J. V. den Bergh, "Practices of knowledge intensive process management: Quantitative insights", *Business Process Management Journal*, vol. 19, no. 3, pp. 515–534, 2013, eprint: https://doi.org/10.1108/14637151311319932, DOI: 10.1108/14637151311319932.

[8] T. H. Davenport and L. Prusak, *Working knowledge: How organizations manage what they know*. Harvard Business Press, 1998.

[9] G. Hube, *Beitrag zur Beschreibung und Analyse von Wissensarbeit*. 2005.

[10] T. H. Davenport, *Thinking for a living: How to get better performances and results from knowledge workers*. Harvard Business Press, 2005.

[11] M. Reichert. (Oct. 2017). Business process management, [Online]. Available: https://www.uni-ulm.de/en/in/iui-dbis/teaching/veranstaltungen/saps/business-process-management/ (Accessed on: May 28, 2018).

[12] N. Mundbrod. (Nov. 2017). Procollab - process-aware support for collaborative knowledge workers, [Online]. Available: https://www.uni-ulm.de/cn/in/iui-dbis/forschung/laufende-projekte/procollab/.

[13] N. Mundbrod and M. Reichert, "Flexible task management support for knowledge-intensive processes", in *21st IEEE Int'l Enterprise Distributed Object Computing Conference (EDOC 2017)*, IEEE, 2017, pp. 95–102, [Online]. Available: http://dbis.eprints.uni-ulm.de/1542/.

[14] T. Chow and D.-B. Cao, "A survey study of critical success factors in agile software projects", *Journal of Systems and Software*, vol. 81, no. 6, pp. 961 –971, 2008, Agile Product Line Engineering, ISSN: 0164-1212, DOI: https://doi.org/10.1016/j.jss.2007.08.020.

[15] Active Collab LLC (US), Norfolk, VA, USA. (2018). Project management tool | activecollab, [Online]. Available: https://activecollab.com/ (Accessed on: Jul. 22, 2018).

[16] Active Collab LLC (US), Norfolk, VA, USA. (2018). Upcoming features roadmap | activecollab, [Online]. Available: https://activecollab.com/roadmap (Accessed on: Jul. 22, 2018).

[17] Asana, San Francisco, CA, USA. (2018). Use asana to track your team's work & manage projects · asana, [Online]. Available: https://asana.com/ (Accessed on: Jul. 22, 2018).

[18] Basecamp LLC (US), Chicago, IL, USA. (2018). Basecamp: Project management & team communication software, [Online]. Available: https://basecamp.com/ (Accessed on: Jul. 30, 2018).

[19] Basecamp LLC (US), Chicago, IL, USA. (2018). Basecamp 3 help, [Online]. Available: https://3.basecamp-help.com/ (Accessed on: Jul. 31, 2018).

[20] Basecamp LLC (US), Chicago, IL, USA. (2018). To-dos - basecamp 3 help, [Online]. Available: https://3.basecamp-help.com/article/48-to-dos#group-to-dos (Accessed on: Jul. 30, 2018).

[21] Wrike Inc., San Jose, CA, USA. (2018). Your online project management software - wrike, [Online]. Available: https://www.wrike.com/ (Accessed on: Jul. 31, 2018).

[22] Wrike Inc., San Jose, CA, USA. (2018). Custom statuses and workflows, [Online]. Available: https://help.wrike.com/hc/en-us/articles/210322785-Custom-Statuses-and-Workflows (Accessed on: Jul. 31, 2018).

[23] M. Pesic, H. Schonenberg, and W. M. P. van der Aalst, "Declare: Full support for loosely-structured processes", in *11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007)*, 2007, pp. 287–287, DOI: 10.1109/EDOC.2007.14.

[24] M. Pesic, H. Schonenberg, and W. M. van Der Aalst, "Declare demo: A constraint-based workflow management system.", *BPM (Demos)*, vol. 9, 2009.

[25] W. M. P. van der Aalst, M. Pesic, and H. Schonenberg, "Declarative workflows: Balancing between flexibility and support", *Computer Science - Research and Development*, vol. 23, no. 2, pp. 99–113, 2009, ISSN: 0949-2925, DOI: 10.1007/s00450-009-0057-9.

[26] M. Montali, M. Pesic, W. M.P.v. d. Aalst, F. Chesani, P. Mello, and S. Storari, “Declarative specification and verification of service choreographiess”, *ACM Trans. Web*, vol. 4, no. 1, 3:1–3:62, Jan. 2010, ISSN: 1559-1131, DOI: 10.1145/1658373.1658376.

[27] J. Santiago and D. Magallon, *Critical path method*, 2009.

[28] J. E. Kelley Jr and M. R. Walker, “Critical-path planning and scheduling”, in *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference*, ser. IRE-AIEE-ACM ’59 (Eastern), Boston, Massachusetts: ACM, 1959, pp. 160–173, DOI: 10.1145/1460299.1460318.

[29] J. D. Wiest, “Precedence diagramming method: Some unusual characteristics and their implications for project managers”, *Journal of Operations Management*, vol. 1, no. 3, pp. 121 –130, 1981, ISSN: 0272-6963, DOI: https://doi.org/10.1016/0272-6963(81)90015-2.

[30] A. Lanz, B. Weber, and M. Reichert, “Time patterns for process-aware information systems: A pattern-based analysis - revised version”, University of Ulm, Ulm, Technical Report, 2009, [Online]. Available: http://dbis.eprints.uni-ulm.de/648/.

[31] E. A. Moore and P. Abrari, “Rule engine”, pat. US7565642B2, US Patent 7,565,642, 2009.

[32] Red Hat Inc. (US). (2018). Drools - business rules management system, [Online]. Available: https://www.drools.org/ (Accessed on: May 14, 2018).

[33] M. Salatino, M. De Maio, and E. Aliverti, *Mastering jboss drools 6*. Packt Publishing Ltd, 2016.

[34] E. Serral, P. Sernani, and F. Dalpiaz, “Personalized adaptation in pervasive systems via non-functional requirements”, *Journal of Ambient Intelligence and Humanized Computing*, pp. 1–15, 2017.

[35] M. Reichert and B. Weber, *Enabling flexibility in process-aware information systems: Challenges, methods, technologies*. Springer Science & Business Media, 2012.

[36] S. Mertens, F. Gailly, and G. Poels, “Towards a decision-aware declarative process modeling language for knowledge-intensive processes”, *Expert Systems with Applications*, vol. 87, pp. 316 –334, 2017, ISSN: 0957-4174, DOI: https://doi.org/10.1016/j.eswa.2017.06.024.

[37] D. Janzen and H. Saiedian, “Test-driven development concepts, taxonomy, and future direction”, *Computer*, vol. 38, pp. 43–50, 2005.

[38] J. Haleby. (2018). Rest assured, [Online]. Available: http://rest-assured.io/ (Accessed on: Apr. 17, 2018).

[39] Oracle. (2017). Introduction to Contexts and Dependency Injection for Java EE, [Online]. Available: https://javaee.github.io/tutorial/cdi-basic.html (Accessed on: Jul. 4, 2018).

Name: André Lang Matriculation number: 937235

Honesty disclaimer

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, ...

André Lang