Supporting Implicit and Explicit Coordination in Software-Intensive Systems Engineering

Thesis submitted to obtain the Venia Docendi in the area of
Applied Computer Science
Johannes Kepler University Linz, Austria
Abstract

One of the central challenges in engineering projects, whether in industrial safety-critical domains, less regulated software systems, or open-source development environments, is effective work coordination. Engineers must navigate complex processes, identify relevant tasks, and understand deviations, all while coordinating with team members. In regulated domains, adherence to rigorous processes is mandated by regulations, but identifying relevant processes and ensuring adherence remains challenging due to complexity and lack of automation support. In less regulated environments, determining deviations from intended processes is cumbersome. In open-source settings, developers struggle with identifying dependent tasks and coordinating with team members, relying on potentially inaccurate links provided by task management tools.

The primary research gap identified in this thesis pertains to the absence of approaches that allow engineering processes to run in the background without mandating engineers to strictly adhere to predefined procedures, while still offering guidance on process status and progress. Engineers require guidance on task readiness, adherence to quality assurance criteria, and deviations from the process, alongside the flexibility to deviate as necessary. Existing methods either require explicit modeling for deviations or are overly flexible, lacking sufficient guidance. Moreover, most approaches necessitate engineer interaction for process tracking, rather than passive observation and inference of process progress from their activities. Additional research gaps include limitations in software process mining due to high entry barriers and insufficient support for the combination of process metrics and timeline visualization. Task-aware artifact dependency analysis also poses a challenge, with existing tools focusing on individual tasks rather than broader process contexts, highlighting the need for investigating change propagation across tasks.

The overarching theme of the thesis is, hence, to develop mechanisms supporting stakeholders in managing implicit and explicit coordination dependencies, providing timely information, and reducing the effort required to obtain critical information about task readiness, quality assurance compliance, process adherence, and artifact dependencies. The thesis considers engineering artifacts, such as requirements and source code, as well as tasks which encompassing various engineering activities, and aims to explore support mechanisms tai-
lored to different process rigor levels. To this end, the seven publications that make up this cumulative habilitation thesis make three major contributions: (1) supporting engineers in following explicit engineering processes, (2) raising awareness of implicitly followed processes, and (3) understanding implicit coordination dependencies.
I would like to express my deepest gratitude to my family for their unwavering patience, understanding, and support throughout the duration of the research that made this thesis possible. Your encouragement and belief in me sustained me through the long hours of research and writing, and I am profoundly grateful for your love and encouragement.

I am indebted to my co-authors for the diverse collaborations and stimulating discussions, which have played a crucial role in shaping my research journey. I'm especially thankful to Prof. Alexander Egyed for pushing me to put together this habilitation thesis and to Prof. Michael Vierhauser for insightful discussions on organizing this thesis.

I am grateful to my coworkers for their camaraderie, collaboration, and insightful discussions. Your diverse perspectives and shared enthusiasm have enriched my research experience and inspired new ideas.

I would also like to acknowledge my host Prof. Richard Taylor at UC Irvine for discussions and the opportunity to conduct research at such a prestigious university that ultimately led me to move from research on distributed systems to pursue a very productive and fulfilling research line in software engineering.

I would also like to thank our industry partners, especially Thomas Mehofer, for their long-standing collaboration, providing me with access to real-world engineering problems and evaluation opportunities. These practical insights and support have been instrumental in bridging the gap between theory and practice.

I extend my sincere appreciation to the funding agency FWF (Austrian Science Fund) and the State of Upper Austria for their financial support, which made this research possible. Without them, I would have joined the industry many years ago and the research presented in this habilitation thesis would not have happened.

Finally, I wish to express my heartfelt thanks to all individuals who have contributed in various ways to the completion of this thesis. Your support and encouragement have been invaluable, and I am deeply appreciative of your contributions.

Thank you all for being part of this journey.
1 Introduction

2 State of the Art and Research Gap
   2.1 Process-centric software development environments ...................
   2.2 Business Process Management (BPM) ...................................
      2.2.1 Process Compliance Checking/Repairing ......................
   2.3 Engineering Process Inspection and Mining ..........................
      2.3.1 Visual Process Inspection .....................................
   2.4 Mining of Implicit Artifact Dependencies ............................
      2.4.1 Explicit Artifact Dependencies - Traceability ..............
   2.5 Research Gap ....................................................

3 Challenges and Contributions
   3.1 Challenge 1: Supporting Engineers in Following Explicit Engineering Processes
   3.2 Challenge 2: Raising Awareness of Implicitly Followed Processes ........
   3.3 Challenge 3: Understanding Implicit Coordination Dependencies ........
   3.4 Contributions Overview ...........................................
   3.5 Contributions to Challenge 1: Providing Engineering Process Guidance
      3.5.1 P1 - Supporting Quality Assurance with Automated Process-Centric
             Quality Constraints Checking .................................
      3.5.2 P2 - ProCon: an Automated Process-Centric Quality Constraints Check-
             ing Framework ..............................................
   3.6 Contribution to Challenge 2: Supporting Process Inspection ............
      3.6.1 P3 - Using constraint mining to analyze software development processes
      3.6.2 P4 - Process Inspection Support: an Industrial Case Study ........
   3.7 Contribution to Challenge 3: Detecting Implicit Cross-Task Dependencies
      3.7.1 P5 - Does the propagation of artifact changes across tasks reflect work
             dependencies? ..............................................
      3.7.2 P6 - Mining Cross-Task Artifact Dependencies from Developer Interaction
3.7.3 P7 - Do Communities in Developer Interaction Networks align with Subsystem Developer Teams? An Empirical Study of Open Source Systems .................................................. 29
3.8 Related Contributions beyond the Scope .................................................. 30

4 Conclusions and Outlook 32

5 Publications 35

5.1 Supporting Quality Assurance with Automated Process-Centric Quality Constraints Checking .......................................................... 36
5.2 ProCon: An automated process-centric quality constraints checking framework 50
5.3 Using Constraint Mining to Analyze Software Development Processes .... 74
5.4 Process Inspection Support: an Industrial Case Study .......................... 85
5.5 Does the propagation of artifact changes across tasks reflect work dependencies? 96
5.6 Mining Cross-Task Artifact Dependencies from Developer Interactions .... 108
5.7 Do Communities in Developer Interaction Networks align with Subsystem Developer Teams? An Empirical Study of Open Source Systems .......................... 120

Bibliography 132
1 Introduction

Effective and efficient work coordination remains a central challenge in engineering projects aimed at building large, complicated software-intensive systems. Regardless of whether the engineers are organized in open source communities, are part of fast-paced frequently releasing teams, or follow tightly defined processes for building safety-critical systems, they all need to be concerned with becoming aware of process progress, understanding deviations, and identifying who to coordinate with.

In industrial, safety-critical domains, software engineers adhere to rigorous processes defined by regulations, standards, and guidelines. These processes include explicit quality assurance measures, traceability paths, and change management protocols to ensure high-quality engineering artifacts and system safety. However, engineers often struggle to identify relevant processes due to the complexity of multiple process variants and regulations across different departments or teams. Timely feedback on process adherence is lacking, as process definitions are typically not machine-executable, leading to manual verification and little automation support.

Industrial engineering teams working on less regulated software systems also aim to follow processes, albeit processes that are less stringently defined. Similarly, determining to what extent the actual behavior during a project differed from the intended process during retrospective analyses is often cumbersome and requires significant manual effort. Furthermore, differences among companies and departments make it hard to determine useful metrics that measure socio-technical congruence.

In contrast, open-source development environments feature decentralized collaboration, allowing developers to contribute across the system. In doing so, developers face challenges in identifying dependent tasks and dependent developers during task execution (such as bug fixing). They rely on links provided by task management tools to make implicit task dependencies explicit. However, the accuracy and completeness of these links remain questionable; often, maintaining traces is considered tedious. Yet, understanding communication patterns and coordination dynamics beyond the explicit link information is crucial, especially as software quality relies on effective collaboration.

To this end, the overarching theme of this thesis is the investigation and development
of mechanisms that support stakeholders such as engineers, team leads, QA engineers, and process engineers in better managing implicit and explicit coordination dependencies. More specifically, we achieve this by (1) providing information timely and (2) reducing the effort to obtain information on aspects such as whether: are tasks ready to be started, are tasks fulfilling all defined quality assurance constraints, are engineers deviating from the intended process, which other tasks have involved work on the underlying engineering artifact, which other tasks are potentially affected by changes of the underlying engineering artifact.

Throughout the thesis and its publications, we consider any form of document, file, web resource, etc. that engineers use, adapt, or create to produce a high-quality software system as an engineering artifact. Typical examples include requirements, source code, models, documentation, and test cases. Similarly, a task\(^1\) is any activity that engineers engage in that results in a high-quality outcome. Hence tasks include not only requirements elicitation, refinement, modeling, coding, and testing, but also artifact reviews, artifact tracing, retrospections, etc. Tasks are often manifested as issues, work items, or change requests – just to name a few – in issue tracking systems or ALM tools such as Jira, Jama, or Azure Devops Services (these just serve as representative examples). Tasks and their dependencies, however, are often just implicitly available in the heads of the engineers as tacit knowledge. The degree of explicit task representation is usually related to the degree of rigorously defined engineering processes. Hence the exemplary settings outlined above require different support mechanisms – mechanisms explored in this thesis.

This cumulative thesis is structured as follows. In the next chapter, we will provide a brief overview of the various research areas that approach the challenges of implicit and explicit coordination from different perspectives. The research gap identified in that chapter 2 is then mapped to three concrete challenges in Chapter 3 together with an outline of how the contributions as part of this thesis address these challenges. Subsequently Chapter 4 provides an outlook on future research directions and open questions. Chapter 5 provides the actual copies of the individual publications.

\(^1\)Often also denoted a step.
2 State of the Art and Research Gap

Our approaches, models, and techniques developed to support implicit and explicit coordination relate to several diverse research areas:

1. Process-Centric Software Development Environments with respect to modeling, monitoring, and enactment of engineering processes.
2. Business Process Management with respect to compliance checking, process mining, and inspection.
3. Mining of artifact changes to determine implicit artifact dependencies.
4. Traceability with respect to managing explicit artifact dependencies.

2.1 Process-centric software development environments

Process-centric software development environments (PCSDE) have received significant attention in the ’90s. We discuss an exemplary selection below, for a detailed review see [1] or [2]. Step-centric modeling and active execution frameworks such as Process Weaver [3], SPADE [4], Serendipity [5], EvE [6], or PRIME [7] determine which steps may be done at any given moment, automatically executing them where possible. While such research supports detailed guidance, deviations from the prescribed process is not well supported. Approaches such as Shamus [8], PROSYT [9], or Merlin [10] specify for engineering artifacts which actions and conditions are available, and enforce their correct order – yet, without prescribing an overall step-based engineering process. Often the supported artifacts are limited to files and folders. Systems such as MARVEL [11], OIKOS [12], or EPOS [13] utilize Event Condition Action (ECA) rules or pre- and post-conditions, thereby providing significant freedom of action to the engineer but offering limited guidance.

The approaches described so far have the implicit assumption that engineers primarily interact with the PCSDE for executing work. We aim to remain in the background, with engineers staying in their tools except for confirming QA constraint fulfillment. Provence [14] has a similar goal, maintaining a process view from artifact change events. It’s, however, limited to events from the file system, relying on moving files to dedicated folders to signal process-meaningful events. It also remains unaware of trace links between artifacts.
More recent work focuses on specific aspects of the engineering life cycle rather than general-purpose processes. DevOpsML [15] aims at reducing the effort to describe continuous integration and deployment processes. Amalfitano et al. [16] aim to fully automate the execution of the testing process and to automatically generate appropriate traceability links. Similarly, Hebig et al. [17] investigate how various software design and code artifacts dependencies emerge from MDE activities. When involving human steps, approaches often assume predefined process models and rigorous tool integration. Kedji et al. provide a collaboration-centric development process model and corresponding DSL [18]. At a micro-level, Zhao et al. propose Little-JIL for describing fine-grained steps involved in refactoring [19] and to help developers track artifact dependencies during rework [20].

A few approaches on general-purpose process modeling and execution (e.g., [21, 22, 23, 24]) focus on step-centric languages such as SPEM and BPMN, which both imply active execution where engineers cannot deviate from the prescribed process.

### 2.2 Business Process Management (BPM)

The BPM research discipline focuses on improving business processes within an organization. BPM involves the analysis, design, implementation, monitoring, and optimization of business processes to enhance efficiency, effectiveness, and adaptability. As such, its approaches and techniques cannot all be directly applied to the software engineering domain.

Software development processes are highly dynamic and iterative, especially in Agile and DevOps environments. Unlike traditional business processes that may have well-defined sequences of steps, software development often involves frequent changes, iterations, and unpredictability. This is because software development is a creative and problem-solving endeavor that involves collaboration, experimentation, and innovation. Unlike many traditional business processes, which are often repetitive and routine, software development requires developers to adapt to changing requirements, explore new technologies, and find novel solutions to complex problems. Note that while especially in regulation-centric fields, software engineering processes are specified with a clear sequence of steps, this sequence mainly describes the ideal flow without any interruptions or unforeseen situations. Processes are primarily established to guide developers. Yet, two distinct BPM research lines are related to software engineering within the scope of this thesis: (1) process checking/repairing (discussed below) and (2) process mining applied to software engineering activities (discussed in Chapter 2.3 below).
2.2.1 Process Compliance Checking/Repairing

In the business process modeling domain, significant related work focuses on formally verifying processes [25] rather than attempting to fix them. Fixing is limited to achieving sound process models but does not apply to instances as we aim for. The few, recent approaches that address inconsistencies and their repair exhibit limited expressiveness for specifying constraints: LTL for expressing task and event constraints [26, 27] or Mixed-Integer Programming for determining runtime compliance of task and resource allocation [28]. Business process compliance checking approaches determine whether complex sequences of events and/or their timing violate particular constraints. Ly et al. analyze frameworks for compliance monitoring [29] and highlight that the investigated frameworks have little or no inherent support for referencing data beyond the properties available in the respective events (hence no access to the actual artifact details and their traces/relations to other artifacts). They also show that hardly any approach supports proactive violation detection, the ability to continue monitoring after a violation, or root cause analysis in a manner useful for software engineering. Also, recent work such as [30] or [31], lacks this crucial support for defining constraints on artifact details.

Notably, the processes studied and used for evaluation in the business process management or information systems domain exhibit complex decision-making about which task to do next, or which task must not be done, and how much time between tasks may pass. Evaluation domains thus often include administrative processes, medical processes, or legal processes but virtually never software engineering processes. In the software engineering domain, processes are simpler but instead require a focus on keeping artifacts consistent with each other. Hence structural (i.e., data-centric) constraints are required which our approach checks proactively, subsequently highlighting that they are not yet fulfilled. Often the necessary guidance is not so much about which task to do next, but when to do it.

In comparison to dedicated software engineering process environments introduced above, general (business) process support, as provided by enterprise tools like SAP, is not applicable in software engineering environments as such support rigidly controls what steps may be worked on without any possibility of deviation.

2.3 Engineering Process Inspection and Mining

Process mining techniques have been applied to software engineering artifacts: most often process events logs comprising commit information from source code version repositories, but also from other implicit process support tools such as email lists and bug trackers. Hence,
we are not considering traditional constraint mining work targeted at independently exist-
ing step-centric business documents, requests, and case files (in contrast to interdependent
engineering artifacts).

Process mining is central to understanding how defined engineering processes are actu-
ally lived, i.e., to understand to what extent engineers deviate from them. Explicit processes,
however, assume control of the process over tools and engineers, which greatly limits freedom,
respectively forces engineers to work outside the process to handle unforeseen situations and
optimizations not foreseen by the process. Diebold and Scherr [32] show that in industrial
practice the majority of processes, therefore, focus on description rather than using formal
notations or models. Organizations tend to apply semi-formal process descriptions contain-
ing different graphical, table-based, or structured-text elements for representation. Process
constraint mining thus becomes a necessity to obtain information on how much the actual
process really matches the official process, highlights process improvement potential, and
serves as input to run-time constraint monitoring, which provides rapid feedback to process
stakeholders.

Poncin et al. [33, 34] introduce the Framework for Analyzing Software Repositories (FRASR)
for combining data from source code repositories, email lists, and bug trackers. They subse-
quently utilize the ProM process mining framework for obtaining insights such as classifying
developers in open source software projects to roles such as project leader, core member,
peripheral developer, bug fixer, or reader. They also analyzed the typical transitions between
bug report states on Bugzilla. The main difference to our approach is that with FRAS-
R/ProM, the engineer has to define what type of constraints/relations to look for and what
the relevant properties of the observations are. Rubin et al. [35] suggest to extract soft-
ware engineering processes from source code repository information. Their approach relies
on classifying artifacts into meaningful categories, e.g., README, CONFIG, or SRC. Maggi
et al. [36] propose to mine declarative process models from events logs. Their DECLARE
approach produces a set of LTL constraints. Gupta et al. [37] conducted process mining
across an issue-tracking system, a code review system, and a version control system. They
map events from these systems into a single process (based on states) and determine tran-
sition occurrences. Based on this annotated transition diagram, they analyze the bug-fixing
process from reporting to resolution to discover bottlenecks, deviations from the intended
process, joint activities, and work handover. Similarly, Akbarinasaji et al. [38] mine a bug
report’s life-cycle for predicting bug-fixing duration. Bala et al. [39, 40] propose an approach
for mining GANTT charts from source code commit history. It requires extensive, explicit
mapping of commits to activities.

Early work on software process mining applied diverse techniques such as mining a petri-net
from versioning logs [41], generating a finite state machine [42], using probabilistic relational modeling [43], or probabilistic event analysis [44]. All these approaches assume that events (e.g., from a commit) adhere to a well-defined, a-priori-known set of activities. Later approaches focusing on mining continuously evolving processes similarly rely on explicit activity types [45, 46, 47].

2.3.1 Visual Process Inspection

Mining approaches that focus on processes are often complemented with visual inspection of individual activities to better understand sources and reasons for deviation. Especially as issue trackers have become an important tool for teams to coordinate their work, managing the increased number of issues, however, has become a challenge [48] that multiple researchers aim to address.

Luijten et al. [49] introduced a tool to generate three different views that enable assessment of the issue handling process: a high-level (Issue Churn View), a quantitative (Issue Risk Profiles) and a detailed life-cycle (Issue Lifecycle View) view. Knab et al. [50] visualize the duration of a process step (submitted, in_analysis, in_resolution, in_evaluation) with a pie chart and provide a state transition view for problem reports.

Sarma et al. [51] proposed Tesseract, a socio-technical dependency browser that enables exploration of relationships between artifacts, developers, bugs, and communications, for example highlighting developers that are modifying interdependent code but are not communicating with each other.

Dal Sassc and Lanza [52] implemented in*Bug, a web-based software visual analytics platform. Extracting data from bug tracking systems, different panels describe high-level information such as duration (as a horizontal stacked bar chart) and status of bugs as well as fine-grained views describing changes to a bug report’s properties.

Similarly, D’Ambros et al. [53] focus on becoming aware of critical issues. Their “Bug Watch” visualization helps to understand the various phases that it traversed. They note that the criticality of a bug is not only dependent on its severity and priority but also on its life cycle. Frequently opened bugs indicate deeper problems.

Halversion et al. [54] describe problematic patterns of change management for example recurrent loops (e.g. repeatedly resolving and reopening or reassigning) or unattended issues (when an issue remains too long without resolution).

Tüzün et al. [55] describe their progress towards a unified project monitoring solution based on the Essence language and kernel. Also, Brandt et al. [56] build on the Essence framework for project state visualization but focus on a Kanban-style visualization rather
than metrics and issue history.

**2.4 Mining of Implicit Artifact Dependencies**

The approaches discussed above focus on identifying and analyzing dependencies between tasks (in the broader sense). Here, we provide an overview of work that focuses on implicit change dependencies among artifacts as a source for identifying coordination needs. The following approaches primarily scope their analysis to artifact changes within a single (often implicit) task.

Prior investigations of artifact change propagation studied primarily the logical coupling between artifacts, i.e. which artifacts tend to co-evolve [57, 58, 59, 60] and not the links among tasks. These approaches observe which artifacts frequently occur in the same commit (or in commits in temporal proximity) independent of the task these commits belong to. Few approaches analyze control and data flow among code artifacts [61], mine association rules from software revision histories [62, 63], or utilize a variability model to detect the impact within product families [64].

Several researchers consider developers’ interaction histories to augment traces among logically coupled source code artifacts [65, 66, 67]. Kostadin et al. [68] use low-level IDE interaction to detect the hidden behavior of developers. Bantelay et al. [65] combine interaction histories and commit data to improve the detection of evolutionary coupling between artifacts. Proksch et al. [69] use developer activities in the IDE with context information, such as source-code snapshots for change events to study developer behavior. These approaches aim to find traces among code artifacts without considering contextual information such as the task the developer is working on. However, Wiese et al. [70] apply contextual information collected from tasks, developers’ communication, and commit data to capture the change patterns of artifacts. They use this contextual information to improve the artifact co-change prediction.

Several task-centric approaches consider fine-grained developer interactions but restrict analysis to interactions within a task without considering the relations to other tasks. Kersten and Murphy [71] introduce the Mylyn tool for determining which code artifacts are relevant for a particular development task.

Hipikat [72] supports the developer in retrieving relevant artifacts from the project’s overall history. It considers documents, tasks, commits, messages, and artifact changes but not the detailed engineering interaction history. A tool that captures the interaction that occurred in a particular file is HeatMap [73]. Another tool, Wolf [74] extracts artifact ownership and changes from source code repositories and generates traces between artifacts and engineers.
The tool provides an organizational view for managers and an individual view for developers to support impact analysis activities. However, such tools focus on the relation between artifacts and tasks but not necessarily on the dependencies of artifacts across tasks.

### 2.4.1 Explicit Artifact Dependencies - Traceability

In safety-critical systems or regulation-centric domains (e.g., aerospace, automotive, and robotics), the software development process needs to produce traces between various artifacts. The need for such traces is to validate/demonstrate that the system under development has expected quality [75, 76, 77]. To this end, utilizing implicit artifact dependencies is insufficient as these tend to be too inaccurate.

Several researchers have proposed techniques for continuously assessing and maintaining software traceability [78]. Event-Based Traceability (EBT) uses a publish-subscribe model to notify developers when trace links need to be updated [79] while Rempel et al. proposed an automated traceability assessment approach for continuously assessing the compliance of traceability to regulations in certified products [80, 81]. These approaches are orthogonal to our work as they are process-unaware, and hence provide little to no guidance for which step in a process a trace link must be available. Furthermore, in this work, we assume engineers have chosen a suitable traceability strategy [82] and assessed that the resulting traceability information model (supported by flexible traceability management tools such as Capra [83]) also conforms to the relevant guidelines [84].

### 2.5 Research Gap

We identified as the primary research gap the lack of approaches for executing engineering processes in the background without forcing engineers to precisely follow the prescribed process while still providing guidance on the process status and progress. Engineers need the best of both worlds: (1) guidance to inform them what tasks are ready to be done, fulfill quality assurance criteria, and where deviations have occurred or would occur — which only can be achieved through a process model — and (2) the ability to flexibly deviate from the process as needed while still being able to receive guidance.

Existing approaches either need to explicitly model the possibility of deviating or are too flexible (hence not providing sufficient guidance). Additionally, most approaches require engineers to interact with the process environment in order to track the process progress rather than a process environment passively observing the engineers’ activities in their tools and inferring process progress in the background from these activities. The few existing
process support approaches that allow the engineer to deviate from the process do not offer any guidance on what are the concrete various alternatives (i.e., actions) that would bring the process back into a consistent state. They neither provide such guidance for ongoing work while the process is in a deviating state. Also, contemporary traceability support approaches provide little to no guidance for which step in a process a trace link must be available as these approaches remain process-unaware.

We identified additional research gaps for process inspection and mining. State-of-the-art software process mining requires a-priori data labeling such as classifying artifacts into meaningful categories and/or mapping events to activity types. Hence, the entry barrier to applying mining algorithms is somewhat high and thus establishes a barrier to exploratory investigation of how the actual processes manifest. It thereby decreases the chances of detecting dependency types process designers didn’t anticipate. Similarly, we didn’t find approaches that allowed a combination of (flexibly selectable) process metrics and timeline visualization.

Finally, we identified task-aware artifact dependency analysis as the third main research gap. The analyzed tools and approaches offer developer assistance independent of tasks or focus on one single task context (but not the overall process context of that task), i.e., the underlying conceptual models lack relations between tasks. Hence we determined that there is a need to investigate change propagation across tasks to highlight coordination needs among tasks that are not explicitly linked.

Overall, an explicit representation of work (i.e., tasks, steps) including awareness of explicit (or rendered explicit) dependencies to engineering artifacts and among engineering artifacts is the foundation for providing engineering guidance and process improvement activities.
3 Challenges and Contributions

In this chapter, we present the three related challenges to which this habilitation thesis makes contributions: (1) supporting engineers in following explicit engineering processes, (2) raising awareness of implicitly followed processes, and (3) understanding implicit coordination dependencies. Together these three challenges cover the spectrum from loosely/non-regulated software engineering environments to heavily regulated engineering environments. Figure 3.1 displays how the individual seven contributions align with the challenge spectrum and which Design Science Research (DSR) activities we conducted in the scope of addressing the challenges. Having introduced the challenges, we then provide an overview of how our contributions address these challenges. Copies of the actual contribution publications are embedded in Chapter 5.

<table>
<thead>
<tr>
<th>Highly-regulated Development Environments</th>
<th>Loosely/Non-regulated Development Environments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supporting Engineers in Following Explicit Engineering Processes</td>
<td>Raising Awareness of Implicitly Followed Processes</td>
</tr>
<tr>
<td>Understanding Implicit Coordination Dependencies</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Contributions</th>
<th>Challenges</th>
</tr>
</thead>
<tbody>
<tr>
<td>P3: “Using constraint mining to analyze software development processes” (International Conference on Software and Systems Processes, ICSSP 2018)</td>
<td>Raising Awareness of Implicitly Followed Processes</td>
</tr>
<tr>
<td>P5: “Does the propagation of artifact changes across tasks reflect work dependencies?” (International Conference on Software Engineering, ICSE 2018)</td>
<td>Understanding Implicit Coordination Dependencies</td>
</tr>
<tr>
<td>P2: “ProCon: An automated process-centric quality constraints checking framework” (Journal of Systems and Software, JSS 2023)</td>
<td>Supporting Engineers in Following Explicit Engineering Processes</td>
</tr>
<tr>
<td>P6: “Mining Cross-Task Artifact Dependencies from Developer Interactions” (International Conference on Software Analysis, Evolution, and Reengineering, SANER 2019)</td>
<td>Understanding Implicit Coordination Dependencies</td>
</tr>
</tbody>
</table>

| Design Science Research Methods | Prototyping Case Study User Experiments Quantitative Evaluation Interviews | Prototyping Case Study Qualitative Evaluation Interviews | Prototyping Qualitative Evaluation Quantitative Evaluation Case Study |

Figure 3.1: Aligning Contributions with Challenges Overview
3 Challenges and Contributions

3.1 Challenge 1: Supporting Engineers in Following Explicit Engineering Processes

Software engineering organizations in safety-critical domains require rigorous processes for their engineers to follow during software and system development to realize the benefits, such as high-quality engineering artifacts [85]. To this end, processes include explicit software quality assurance measures (QA) such as the creation of traces between artifacts [86, 84], conducting artifact reviews, test planning, change management, and many more. Regulations, standards, and guidelines specify the required final traceability paths as evidence for system safety, for example, the U.S. Food and Drug Administration (FDA) principles in the medical domain [87, 88], DO-178C/ED-12C for airborne systems [89], and Automotive SPICE (ASPICE) [90] or ISO 26262[91, 92] in the automotive industry. It is up to the organizations, however, to define suitable processes and operationalize them.

One challenge for engineers in these settings is adhering to the process [93], as there is rarely a one-size-fits-all process within an organization. Processes evolve and improve continuously, hence multiple versions of the same process will co-exist within an organization. Different customers demand following slightly different process variants (and traceability paths). Individual departments or teams responsible for different subsystems might have to adhere to different regulations.

Engineers at our industry partners in the automotive and air-traffic control domain stated that they find it hard to identify which precise process is relevant to their current work context. They reported being stressed about potentially missing important steps mandated by quality assurance. Especially, the challenge to correctly provide traces between engineering artifacts — a sub-problem of this grander challenge — has been identified as a common concern in the automotive industry [94].

Understanding the process definitions and their implications is a daunting task, especially for newcomers. When engineers confirm with colleagues what quality assurance activities they need to conduct—and when—to fulfill their process, they need to avoid accidentally obtaining inapplicable feedback from someone familiar with a different process variant. Engineers in our interviews expressed their desire to have immediate feedback on their specific process instance to understand which steps are ready to start, what actions are missing to complete their step, and when rework has happened.

In industry, however, process definitions are typically not described in a machine executable format [32]. As a consequence, there is little to no automation support for the developer to check whether processes are followed, the extent to which deviations occur during development, or what needs to be done (next) to adhere to the process.
The current practice is that QA Engineers need to conduct countless, tedious, often mind-numbing checks that involve (manually) navigating across diverse artifacts and tools to ensure that the required constraints are fulfilled at the right process step. These checks are error-prone and rarely conducted in time to provide immediate feedback to developers. We observed that when quality checks are performed in batches for efficiency reasons towards the end of the development cycle, developers may only receive feedback as late as 6-12 weeks after completing their work. Remediating problems late in the process interrupts developers who may have already moved on to other steps or projects, causing disruptions and extra effort as they need to re-understand their past work context.

To better convey these challenges we first provide an insight into the level of detail processes entail that are found in the wild. We provide in Figure 3.3 an excerpt of the requirement refinement process similar\(^1\) to one at one of our industry partners designed to be in compliance with Automotive Spice. It covers the highlighted part of the V-model in Figure 3.2 and is itself sub-structured into two additional subprocesses provided in Figure 3.5 and Figure 3.4, respectively.

For a concrete example, we focus on the sub-process of Analyse System Requirements depicted in Figure 3.5. Suppose Alice needs to create, respectively update, an existing Impact Analysis Documentation to ensure that environmental effects are clearly identified and documented. This step typically has the User Requirements Set (URS), and System Requirements Specification (SRS) as input. URSs and SRSs need to be in state “Approved” and ”Draft”, respectively to signal to Alice that these artifacts are ready to be worked on.

The challenges here are manifold. First, Alice needs to recall exactly what are the applicable pre-conditions she would need to ensure before she starts her work. Similarly, she needs to understand that for her particular process context, she then needs to trace the system requirements to the user requirements.

Second, Alice might need to identify the extent to which her step’s pre-conditions are not fulfilled yet to assess the risks of rework should she decide to start early. Realizing that most User Requirements have been approved but one is not in the right state yet could present an acceptable risk. Obtaining such information requires time, especially when multiple other engineers are responsible for those artifacts.

Third, a customer demands a change with implications on the SRS. The change is potentially relevant to the Impact Analysis Documentation and needs to be communicated to Alice on time. Understanding the impact of changes requires in-depth knowledge of the process

\(^1\)Due to confidentiality concerns the precise process cannot be disclosed, but the presented one is equivalent in complexity and granularity.
Figure 3.2: Traceability Requirements in Automotive SPICE (annotated copy of Automotive SPICE 3.0 Figure D.4)

Figure 3.3: System Analysis Process.
3 Challenges and Contributions

Figure 3.4: System Requirements Joint Review subprocess.

Figure 3.5: Analyse System Requirements subprocess.
in general, and the artifacts and their dependencies of the underlying process instance in particular.

As the scenario above exemplifies, in most software engineering environments, one cannot expect engineers to precisely follow the prescribed process definition. Various factors such as time pressure, unclear or missing information, or changing customer expectations, cause rework and make iterations necessary. Modeling all such possible “deviations” from an ideal process is often impractical and hence not done. The process, however, has an important guiding purpose, while additional QA constraints define properties of the engineering artifacts and their relations. The key aspect here is that strictly following the process is as detrimental to software quality as largely ignoring it. Hence, the challenge is having less detailed, but nevertheless, informative processes that are mapped to tools (in which they are executed), while tolerating deviations.

3.2 Challenge 2: Raising Awareness of Implicitly Followed Processes

Instead of dedicated process “enforcement” engineers, organizations often rely on the engineers’ own discipline and understanding of the benefits of following the process to ensure that the engineers indeed follow the prescribed process. To this end, most software development organizations nowadays use issue-tracking systems, which offer process guidance and support for process monitoring via basic tracking capabilities. However, detecting tasks that take unusually long or deviate in other ways from the expected process still remains difficult.

When defining, analyzing, and improving software processes it has turned out useful to distinguish several views [95]:

- the perceived process (what developers think they do), i.e., their subjective perspective on the process based on their personal and daily work experience;

- the actual process (what developers really do), i.e., how they execute the process, e.g., as determined through observation; and

- the official process (what developers are supposed to do), i.e., the process model defined by an organization.

Obviously, these views will commonly differ in practice. For instance, processes are frequently changed due to current project needs. Further, they typically vary in different projects and teams of an organization. Thus, when improving software development processes, understanding the perceived, actual, and official processes provides the foundation for defining
3 Challenges and Contributions

Figure 3.6: Process states and transitions for issue type Task.

...an initial process, which can be continuously improved towards a target process based on experience and observation [95]. In this context, observing processes is desirable but very challenging to detect deviations from the official processes early, in order to allow teams to react accordingly. Currently, process monitoring is often based on the official process available before the start of a project (e.g., as defined in an issue-tracking tool). Since the actual behavior in a project may differ significantly, using the official process as a reference for comparison is often not feasible.

In these settings, processes are often only partially defined in the form of which work artifacts in an issue tracking system such as Jira can go through which transitions. Additional constraints such as which role should execute an activity, however, are defined only informally outside of Jira.

For instance, Figure 3.6 shows an excerpt of the allowed process states and transitions for the issue type Task at our industry partner ACME-RA (a company active in website development for recreational activities). Once a new issue is created its initial state is Open. The state changes to In Development as soon as the work on the task starts. After finishing the task the developer changes the state to Ready for Review. A tester then picks up the task and assesses the issue before changing the state to Reviewed. If a task is regarded as finished, its state changes to Resolved (one of the allowed end states), either after review or immediately after development. If required, testing can be performed after resolving the
task (state In Testing). Both development and testing can be suspended (states Suspended Development and Suspended Test), e.g., if additional resources are required, and then resumed when these resources become available.

In case of delays, a difficult development task may remain in the state In Development for an unusually long time. Role violations occur if a state is changed by an unauthorized user, e.g., if a task that is Ready for Review is not Reviewed by a tester. Finally, combined violations may happen: for instance, moving an issue from Suspended Test back to In Testing has to be done within a reasonable time to avoid delays caused by blocking the testing task. While most tracking systems enforce process tracking via state transitions they do not support transition conditions, hence any user with write access may update the issue state without time limitations within the bounds of the defined transitions. Defining and monitoring process constraints, e.g., on delays and role violations, would appear to be desirable but becomes infeasible in highly dynamic settings due to the high number of projects, issue types, and instantiated issues in organizations. Additionally, constraints will need to vary for different types of issues (e.g., a bug fix will usually require less time than implementing a new feature) as well as different projects and teams (e.g., some projects may have a separate quality assurance (QA) group, while in others the developers are responsible for testing too).

Ultimately, the challenge here is how to automatically reveal actual process characteristics on the fly to raise awareness to what extent processes are implicitly followed.

3.3 Challenge 3: Understanding Implicit Coordination Dependencies

In engineering environments where developers are more dynamically involved to varying degrees of intensity (e.g., most often encountered in open source software development), process know-how is less prevalent, and consistent linking (i.e., tracing) of engineering artifacts is not commonly found. Hence, implementing a change such as fixing a bug, introducing a new feature, or removing outdated functionality becomes a challenging task. Correctly and completely implementing a change requires the developers to identify all relevant artifacts. In non-trivial systems, a change often requires editing artifacts that are maintained by different developers or different teams. Aside from small, localized bug fixes changes are rarely described and managed by a single task but rather a set of tasks worked on by different developers.

Crucially, a developer needs to know which artifacts under someone else’s control have an impact on her underlying work task: changes to such artifacts then may induce additional
changes, might restrict how to implement a change, and when to do so. In turn, a developer’s artifact changes cascade to other artifacts under someone else’s control. We characterize the situation when changes to artifacts in one task influence (potential) changes to other artifacts in another task as a cross-task dependency. Simple, illustrative examples include fixing a bug in business logic and updating integration tests accordingly, introducing a new field in the database and displaying it on the user interface, or introducing a feature toggle and adding the toggle trigger to the configuration database. From the author’s experience, in many organizations often different teams are responsible for these tasks due to prescribed development processes, required expertise, or organizational structure.

Awareness of cross-task dependencies is especially important as lack thereof is a common cause of incomplete and incorrect change propagation. Developers thus need support in becoming aware of these dependencies, respectively the involved artifacts for simpler coordination of change propagation (i.e., forward and backward impact assessment). Let us investigate an actual example task subset from the Mylyn open-source project. Mylyn [97] allows a developer to connect to a task management tool (such as Bugzilla) for selecting tasks to work on and captures all developer read and write events within the Eclipse IDE. The tasks in our example address different mechanisms for creating a new Mylyn task. Figure 3.7 depicts the links among tasks as of Nov. 14, 2007. The central Task 169426 has links to five tasks. The greyed-out Task 210022 has not been set up yet. Task 209892 (bold) was just created and thus no progress has been made yet. All tasks are in status “open”. As the developer S.P. assigned to Task 209892 commences work, he needs to know where to look for artifacts and their (recent) changes relevant to the realization of his task. Likewise, the developers currently working on the other open tasks need to assess who they should work with and perhaps notify about changes. Yet, in the month before Nov. 14, 2007, 59
developers were accessing \( \sim 1300 \) artifacts in 164 tasks. Attempting to manually understand the relevance of each of those tasks and artifacts is infeasible. Relying only on explicitly created links between tasks will often be insufficient as this could lead to missing important developments in non-linked tasks.

The main challenge here is identifying which artifacts and other work items such as bug reports, stories, etc. are relevant to the work at hand, given that explicit trace information might be incomplete, inaccurate, or potentially completely missing.

### 3.4 Contributions Overview

In this section, we provide an overview of how the seven publications as part of this habilitation thesis address the three challenges introduced above. The publications and this author’s main contributions to them are listed in Table 3.1. The detailed author contributions form signed by each co-author is separately attached to the habilitation thesis package.

Publications P1 and P2 address the challenge of **Supporting Engineers in Following Explicit Engineering Processes by Providing Engineering Process Guidance**. Publications P3 and P4 address the challenge of **Raising Awareness of Implicitly Followed Processes by Supporting Process Inspection**. Finally, publications P5, P6, and P7 address the challenge of **Understanding Implicit Coordination Dependencies by Detecting Implicit Cross-Task Dependencies**.

Our research procedure was heavily influenced by the Design Science Research (DSR) paradigm. DSR focuses on creating solutions to real-world problems through the design and evaluation of artifacts. The key components of Design Science Research include:

**Problem Identification** DSR begins with the identification of a specific problem or opportunity in a real-world context. Especially for the challenges towards medium- and high-regulated development environments, we relied on our year-long interaction with our industry evaluation partners in (but not limited to) the air-traffic-control domain and automotive domain to identify the lack of proper process guidance (P1-P4). For addressing challenges primarily emerging in an open-source development environment (P5-P7) we relied on ternary sources (i.e., problems reported in other research work).

**Design and Development** Researchers then design and develop artifacts, which can be software systems, models, processes, frameworks, or methodologies, aimed at addressing the identified problem. For all our papers (P1-P7) in this habilitation thesis, we heavily relied on prototypes, many of which reached a high maturity level, so that even engineers in the while can comfortably use them. This allowed us to conduct experiments
not just with university students (mimicking beginners or newcomers) but also with actual engineers at our industry partners (P2, P4).

**Evaluation** The artifacts are rigorously evaluated to assess their effectiveness, efficiency, and utility in solving the identified problem. Through the extensive gathering of historical engineering data, we were able to analyze in depth how our algorithms and prototypes would have performed when applied in these actual software development projects (P1-P7). All quantitative analysis was applied to actual real engineering data, no synthetic data was generated for this purpose in any of the seven publications.

**Reflection and Learning** Throughout the research process, researchers reflect on their findings, iterate on the design, and learn from the outcomes to improve both the artifact and the research process itself. From the controlled experiments and subsequent interviews with involved engineers, as well as from discussion of the experiment results with domain experts, we were able to identify areas of improvement as well as potential misuse (P1, P2, P3, P4).

**Contribution to Theory and Practice** DSR aims to contribute not only to practical problem-solving but also to the advancement of theoretical knowledge in the respective domain. For example, in highly regulated development environments, the novel concept and approach of passive process execution provides a novel viewpoint of perceiving processes as a technical mechanism for guidance without the necessity of controlling engineers (P1, P2). For loosely/non-regulated development environments, we provided novel metrics for measuring change propagation and team alignment (P5-P7).

While contributions have a primary targeted challenge and each challenge emerged from a different spectrum part of the engineering environment, the contributions, nevertheless, can be applied in different contexts: e.g. process guidance for open source teams that aim to build safety critical systems and need to enable quick on-boarding of new contributors. Vice versa, interaction and artifact change dependency analysis may support organizations developing safety-critical systems in their analysis of whether traces are complete and correct. These application areas, however, have not been investigated in the scope of this thesis.

### 3.5 Contributions to Challenge 1: Providing Engineering Process Guidance

Over the past decades process execution was always perceived as a mechanism to control and constrain the work of people. Consequently, as engineers require significant amounts of flexibility, rigorous process support for software engineering environments has had only
Table 3.1: Publications grouped by Challenge detailing the author’s contribution.

<table>
<thead>
<tr>
<th>Id</th>
<th>Publication</th>
<th>Contribution</th>
</tr>
</thead>
</table>
| P1 | Supporting Quality Assurance with Automated Process-Centric Quality Constraints Checking. | - Passive Process Engine Concept Development  
- Process Engine Prototype Implementation  
- Engineering Process Data Collection  
- Process Replay Experiment Preparation and Execution  
- Related Work Analysis  
- Result Analysis and Discussion |
- Data Set Extension  
- User Experiment Preparation and Execution  
- Result Analysis and Discussion |
| P3 | Using constraint mining to analyze software development processes. | - Applying Constraint Mining to Engineering Processes Discussion  
- Related Work Analysis  
- Process Data Preparation  
- Domain Expert Interview Preparation and Execution  
- Result Analysis and Discussion |
| P4 | Process Inspection Support: an Industrial Case Study. | - Mining and Visualization Concept Development  
- Experiment Result Analysis  
- Use Case Description  
- Domain Expert Interview Preparation |
| P5 | Does the propagation of artifact changes across tasks reflect work dependencies? | - Cross Task Change Propagation Concept Development  
- Propagation Metric Algorithm Implementation  
- Related Work Analysis  
- Quantitative Data Analysis  
- Qualitative Data Sample Coding and Analysis  
- Result Analysis and Discussion |
| P6 | Mining Cross-Task Artifact Dependencies from Developer Interactions. | - Cross Task Mining Approach Development  
- Prototype Design Decision Discussions  
- Data Provisioning  
- Result Analysis and Discussion |
| P7 | Do Communities in Developer Interaction Networks align with Subsystem Developer Teams? An Empirical Study of Open Source Systems. | - Approach Concept and Design Discussion  
- Network Metric Development  
- Experiment Design  
- Experiment Result Discussion |
limited success. Accurate and timely guidance, however, can only be provided if the current state of the process is known. The primary mechanism for addressing this dilemma is the concept of passive process execution: tracking the state of the engineering process purely via observation of engineering artifact changes without requiring any explicit input from engineers. The result passive process engine, described in the following two publications P1 and P2 directly address the challenge of Supporting Engineers in Following Explicit Engineering Processes by providing feedback on process violations and how to fix them.

3.5.1 P1 - Supporting Quality Assurance with Automated Process-Centric Quality Constraints Checking

Explicitly modeling a software engineering process for controlling the software engineering life-cycle is not new, with a plethora of research dating back until the 90s [3, 4, 98, 99]. The approaches assume that engineers explicitly interact with the process representation to mark steps are complete or to select steps to work on without any possibility of deviating from the prescribed process definition unless there is such a deviation explicitly modeled.

Contribution

In this paper, we present ProCon, a novel approach in stark contrast to prior process support. It is different insofar as that (i) the process is tracked in the background, based on the software engineers’ activities performed in the tools they are using in their daily work rather than requiring engineers to interact with a process engine, (ii) engineers are free to deviate from the process, (iii) engineers may receive guidance even in the presence of deviation, and (iv) ProCon supports control- and dataflow conditions as well as constraints across diverse artifact types and tools. The key novelty is treating (quality) constraints neither as an implicit part of the engineering process model nor as completely disjunct from it. Instead, we propose treating (quality) constraint evaluations as first-class citizens: i.e., as explicit development artifacts that determine process progress.

Evaluation

We evaluated ProCon against two distinct use cases for which we created process specifications and constraints, and implemented connectors for different issue tracking and requirements management tools. The first use case, Dronology [100] – an open source project – represents a more agile, lightweight process, whereas the second one – a safety-critical system in the air traffic management domain – describes a rigid, standardized process with stringent quality assurance criteria. For the latter use case, we obtained a total of around 27,000 change events over 14,000 engineering artifacts from 109 process instances. With ProCon we were able to quickly check quality constraints and demonstrated that engineers at our industry partner
indeed fulfilled quality constraints (upon frequent feedback from QA engineers) but sometimes failed to correctly provide evidence of their work as required by the process. Additionally, we received very positive responses from engineers at our industry partner upon presenting ProCon, with one team lead wishing to have it ready as a product by tomorrow, and a QA engineer joking to be out of work then. QA engineers at our industry partner also used the prototype for writing new QA constraints during a company-internal innovation event to showcase its potential and adaptability.

3.5.2 P2 - ProCon: an Automated Process-Centric Quality Constraints Checking Framework

In this paper, we extended upon the work of contribution P1 mostly with respect to process engineering activities and further validation aspects.

Contribution
We describe in this contribution in more detail the revised process model underlying the ProCon framework. We further describe process engineering activities: i.e., how to model processes and constraints in ProCon, thereby providing a visual, web-based editor. The extended journal version also includes an in-depth discussion of the technical architecture of ProCon and its user interface for engineers to obtain constraint fulfillment feedback.

Evaluation
We extended the pre-existing evaluation with an analysis of a third industrial data set. The focus of this data set was on evaluating whether ProCon can deal with extensive rework and step repetitions. To this end, we demonstrated that ProCon is able to accurately describe the process progress even in the presence of frequent repetition, pausing, or skipping of engineering activities. This is especially relevant in highly iterative development processes that frequently switch between development and testing phases.

We also evaluated the process design aspects in a preliminary controlled experiment with beginners. The results hint at an easy learning curve as beginners were able to write correct rules/constraints within an hour. Beginners made the least mistakes when writing step Transition rules, and found writing DataMapping rules the hardest. QA engineers at our industry partner found adapting existing constraints to new processes without any external support doable in a timely manner.
3.6 Contribution to Challenge 2: Supporting Process Inspection

To improve coordination, to improve processes, one has to know what to measure. Understanding, however, what exactly to measure is difficult as often teams, organizational structures, processes, etc. change over time, and what once was a useful metric, is no longer very meaningful. Quickly obtaining insights into actual process execution, and actual coordination is necessary to take corrective or optimizing actions.

Hence, towards supporting process inspection, contributions P3 and P4 focus on lightweight mechanisms that don’t require much configuration, are quick to set up, provide insights into what are typical process constraints, what development context gave rise to a particular metric outcome, and how these metrics compare across teams. Switching between recurring process execution patterns, visual inspection of individual process timelines, and assessing process metrics in comparison to other teams, allows stakeholders to understand where coordination can be improved.

3.6.1 P3 - Using constraint mining to analyze software development processes

In order to improve processes via better coordination, it is necessary to determine how the actual process (what developers do) differs from the official process (what developers are supposed to do), without having to rely on the perceived process (what developers think they do). Since the actual behavior in a project may differ significantly, using the official process as a reference for comparison is often not feasible. Process mining approaches [33, 35, 101] have thus been proposed to automatically learn constraints characterizing the actual process, which can then be used to monitor and improve development. All these approaches assume that events (e.g., from a commit) adhere to a well-defined, a-priori-known set of activities.

We describe the application of a constraint mining approach [102, 103] developed in earlier research on software monitoring [104] to analyze software development processes. We first extract data from an issue-tracking tool and then reveal process definitions in the form of constraints. Specifically, our approach can extract different types of constraints on event occurrence, timing, and data (as well as combinations of these). The approach presents candidate constraints to users in a domain-specific language [105] and offers a range of filtering and ranking strategies [103] to select the constraints to be used for monitoring.

Contribution

The main contribution of this paper is our mining approach which does not require data labeling such as classifying artifacts into meaningful categories and/or mapping events to activity types (the main distinguishing factor from the state-of-the-art approaches). Hence, the entry barrier to applying our mining algorithm is very low. It focuses on frequently
occurring event sequences, including value constraints as well as duration. This greatly supports the exploratory investigation of how the actual processes manifest as it increases the chances of detecting dependency types process designers didn’t anticipate.

**Evaluation**
We empirically evaluate our constraint mining approach using data from four software development projects at our industry partner ACME-RA. Specifically, we analyze event logs created from Jira logs to uncover constraints of development processes that need to hold during process enactment. We asked a domain expert from our industry partner to analyze the constraint candidates for the largest project to qualitatively assess the usefulness of our constraint mining approach and to assess the criticality of all useful constraints. The domain expert considered 88/127 of the mined constraints useful rated many (47) as highly critical, and made several other positive comments.

### 3.6.2 P4 - Process Inspection Support: an Industrial Case Study

Research over the past decades has shown that organizational factors such as team structure, coordination among engineers, or processes significantly impact software quality and development progress [106]. Engineers working on strongly coupled artifacts tend to require frequent communication to coordinate their engineering efforts [107].

Therefore, in the context of this industrial case study investigated in this paper, our industry partner ACME-RA was interested in measuring and assessing how its organizational structures and issue-handling processes ultimately affect coordination among engineers and timely delivery in order to obtain insights into how and where to focus on improvements. Typically, software process metrics provide such insights and multiple research efforts aim to improve them [51, 49, 52, 72]. Yet, determining which metrics are useful and accurately describe the ongoing development efforts is non-trivial as this differs among companies and often also among departments and groups within the same company.

**Contribution**
We devised an approach of pairing process metrics with visual historical inspection of issues to overcome the limitations of metric inspection without context on the one hand and visualization without guidance on the other hand. Stakeholders such as managers, team leads, or quality assurance engineers inspect metrics (and deviations from expected values) for individual issues and utilize a historical visualization of the affected (and related) issues to obtain insights into the reason for the metric (deviation) and its root cause. We designed the accompanying prototypical tool (*Process Inspector*) to be lightweight and flexible to support easy integration and adaptation of metrics.
**Evaluation**

We conducted semi-structured interviews with four stakeholders (i.e., actual end-users of the prototype): team leaders and group leaders. Each (separate) interview consisted of an introduction of the prototype including an explanation of the metrics, used data, and user interface features. Subsequently, each participant was asked to assess three issues with respect to identifying coordination problems. Participants explicitly highlighted the usefulness of combining metrics and timelines, quickly finding process flaws in a short time, comparing projects, and the prototype’s simple design. They perceived those metrics as most useful that either utilized changes to re-assignment of issue responsibility, state changes, or date changes.

### 3.7 Contribution to Challenge 3: Detecting Implicit Cross-Task Dependencies

Implementing changes in software development, such as fixing bugs or introducing new features, requires developers to identify all relevant artifacts, often maintained by different teams. This awareness of cross-task dependencies is crucial, as often a developer cannot make all required changes themselves, or cannot assess the impact on other parts of the system. Becoming aware of dependencies across tasks helps to identify with which other engineers to coordinate.

To this end, publication P5 empirically investigates to what extent explicit work dependencies match change propagation and to what extent implicit dependencies remain hidden. Based on this study’s insights, publication P6 provides a mining approach to identify artifacts that are involved in multiple tasks and hence need extra awareness to ensure correct and complete change propagation. Finally, publication P6 empirically analyses directly the interaction dependencies of open source developers within subsystems and across subsystems (as a proxy for implicit tasks) to understand whether communication and coordination structures among these teams exist that would facilitate change propagation.

#### 3.7.1 P5 - Does the propagation of artifact changes across tasks reflect work dependencies?

When creating tasks, developers face the challenge of identifying dependent tasks. One would assume that developers use the links offered by task management tools to make the implicit task dependencies explicit—links that may be inaccurate and incomplete at times. They then use the links to identify which developers to notify/invoke about changes and which artifacts
to change. Since identifying relevant engineers and artifacts for change propagation remains a significant problem [108, 109, 110], the question arises when do links actually reflect change propagation needs?

To this end, this paper analyses the temporal relationship between developer reading and changing source code on the one hand, and task links on the other hand, as found in the Mylyn data set. Mylyn is an open-source task management tool for the Eclipse IDE that captures traces of developer interactions (i.e. artifact reads and writes). Mylyn developers use this tool during their work on Mylyn. Mylyn, therefore, serves as the data-gathering tool as well as the system under investigation in this paper.

**Contribution**

The paper provides one of the first analyses of change propagation across tasks. To this end, we introduced novel metrics to measure bi-directional change propagation and conducted a quantitative and qualitative analysis of developer interaction data.

**Evaluation**

We analyzed development metadata from the Mylyn project which details which source code files developers accessed and which they changed during work on an issue. This information was available for 410 issues with 160 links between those issues. We then analyzed to what extent these artifacts were accessed later in another task and investigated what relationship type the two tasks had (if any). We identified a set of seven situations that explain when linked task pairs exhibit change propagation. In three situations developers use links to manage task dependencies that do not entail change propagation. Examples are task synchronization and task decomposition. The other four situations describe distinct artifact-centric task dependencies. Artifact reuse or work continuation dependencies, for example, explain why task pairs exhibit strong change propagation. We additionally identify six motifs that explain why non-linked task pairs exhibit change propagation. Specifically, the motifs highlight that change propagation alone is insufficient to reliably determine dependent (but non-linked) task pairs.

### 3.7.2 P6 - Mining Cross-Task Artifact Dependencies from Developer Interaction

Implementing a change such as fixing a bug, introducing a new feature, or removing outdated functionality is a challenging task. Lack of awareness of cross-task dependencies is a common cause of incomplete and incorrect change propagation. To this end, we introduce an approach for mining cross-task dependencies from developer interactions with engineering artifacts as captured in the IDE. Interactions describe which artifacts a developer has accessed and edited within the scope of a task. From task pairs, we extract re-occurring artifact sequences: the
3 Challenges and Contributions

cross-task dependencies. We further observe the accessed artifacts during live editing in the IDE and apply the dependencies to recommend which other artifacts are potentially affected by the ongoing work.

Contribution
The main contribution is a technique for extracting cross-task dependencies from interaction data. Contemporary approaches derive dependencies from within-task data or data from aggregated tasks of very close temporal proximity. Additionally, we provide a recommender prototype that applies developer interactions in order to suggest a ranked list of affected artifacts.

Evaluation
Similar to contribution P5 above, we obtained real developer interaction data and tasks from the Mylyn project and mined cross-task artifact dependencies in a sliding window over the duration of several years. Our approach lists 50% of correctly recommended artifacts within the top-5 results. The results demonstrate we are able to successfully find not only cross-task dependencies but also provide them to developers in a useful manner.

3.7.3 P7 - Do Communities in Developer Interaction Networks align with Subsystem Developer Teams? An Empirical Study of Open Source Systems

Developers who work on the same task, artifact, or subsystem need to coordinate and thus communicate in order to avoid incomplete change propagation, rework, or duplicate work. We therefore would expect them to form stronger interaction ties [111] than developers working in different subsystems. We hypothesize that these developer interactions give rise to a community structure. Therefore, the goal of this study is to obtain first insights into the alignment of sub-communities within open source projects with the systems’ structure. To this end, we empirically investigate in this paper, in the context of 10 open source projects, how communities emerge and change over time – e.g., how developers join and leave sub-communities – and the extent to which these community patterns match the subsystems’ evolution. Overall, answering this work provides insights into whether subsystems in open source projects represent decoupled work scopes that result only in limited coordination overhead compared to work coordination within a subsystem.

Contribution
The primary contribution of this paper is an empirical study investigating the evolution of developer communities and their alignment with the subsystems’ developer teams. The secondary contributions of this paper are a technique for measuring alignment (i.e., overlap)
between subsystems and developer communities, and a technique for determining developer communities’ evolution.

**Evaluation**
We extracted issues and commit data from Jira and GitHub for 10 open-source projects over a period of multiple years, from which we extracted developer interaction (via issues) and developer contributions (via commits). We find that developer communities change considerably across a project’s lifetime (hence implying that relevant relations between developers change) while subsystem developer teams (SDTs) remain comparatively stable. Overall, the community alignment with SDTs is often low, which implies that developers maintain significant communication ties with developers outside their (subsystem) work scope. We hypothesize that such an interaction network independent from subsystems emerges from the need to remain robust against the disruption of leaving developers and quick onboarding of new members.

### 3.8 Related Contributions beyond the Scope
The work at the core of this habilitation thesis has been inspired by, supported by, and, in turn, given inspiration to a significant number of related research endeavors. These related contributions span across the areas of flexible business processes, describing human collaboration patterns, and supporting engineering artifact traceability.

In the context of process support for small and medium-sized businesses, we explored the first concepts for determining (business) process progress based on incoming business artifacts. This inspired the notion of observing process status in the background:


We also investigated the modeling of human collaboration (i.e., also developer collaboration and coordination) in terms of concepts known in software architecture. The resulting
“human Architecture Description Language” enables us to model how engineers use engineering artifacts as coordination artifacts and the traces between those artifacts that form the essential foundation over which process and quality constraints are defined:

- Christoph Dorn, Richard N. Taylor: Coupling software architecture and human architecture for collaboration-aware system adaptation. in Proceedings of ICSE 2013: 53-62

In further work, we investigated mechanisms to create and maintain high-quality traces among engineering artifacts:

4 Conclusions and Outlook

This introduction provided an overview of the addressed problem statement, identified the research gap, and how we contributed to the research challenges that emerged from that gap.

The addressed problem can be summarized as follows: the challenge of effective and efficient work coordination is central in engineering projects, regardless of whether engineers operate in open-source communities, fast-paced release teams, or safety-critical domains with stringent regulations. In safety-critical domains, engineers adhere to rigorous processes defined by regulations, struggling with the complexity of multiple process variants and the lack of timely feedback on process adherence. Industrial engineering teams, dealing with less regulated software systems, also aim to follow processes but face challenges in determining deviations during retrospective analyses. In contrast, open-source development environments allow decentralized collaboration but pose challenges in identifying task dependencies and maintaining accurate links. Overall, this thesis brings together approaches that focus on mechanisms to support stakeholders in managing coordination dependencies by providing timely information and reducing the effort required to obtain it.

The primary research gap identified is the absence of approaches for executing engineering processes in the background without mandating engineers to strictly adhere to the prescribed process while still offering guidance on process status and progress. Engineers require both guidance on task readiness, quality assurance fulfillment, and deviation identification, facilitated through a process model, as well as the flexibility to deviate from the process while still receiving guidance. Existing approaches either lack explicit deviation modeling or provide excessive flexibility without sufficient guidance. Moreover, they often necessitate engineer interaction with the process environment instead of passively observing activities for progress inference. Contemporary traceability support methods also lack guidance on step-specific trace links as they remain process-unaware. Additional research gaps include limitations in software process mining due to high entry barriers and the absence of combined process metrics and timeline visualization. Furthermore, there’s a need for task-aware artifact dependency analysis to address coordination needs among tasks not explicitly linked, emphasizing the importance of an explicit representation of work and dependencies for effective engineering guidance and process improvement.
We structured the research gap along three challenges and introduced the set of papers that jointly make up a contribution towards addressing each challenge. Publications P1 and P2 address the challenge of **Supporting Engineers in Following Explicit Engineering Processes** by **Providing Engineering Process Guidance**. Publications P3 and P4 address the challenge of **Raising Awareness of Implicitly Followed Processes** by **Supporting Process Inspection**. Finally, publications P5, P6, and P7 address the challenge of **Understanding Implicit Coordination Dependencies** by **Detecting Implicit Cross-Task Dependencies**.

The work described in these papers is just the start of a longer research line that spans from research on engineering support for passively executed processes to in-the-field observations of how guidance is utilized by actual engineers, to more targeted guidance, to additional mechanisms that inspect the content of engineering artifact to enable more precise and timely guidance.

With respect to the design support aspect, investigations are needed on how process and QA engineers can utilize the ProCon framework to specify processes and constraints. Especially the mapping of natural language constraints—as communicated to engineers—to OCL constraints, perhaps even automatically, and the testing and versioning thereof requires dedicated attention. We envision a full-fetched integrated development environment (IDE) for this purpose as process and constraints require the same rigor in requirements specification (what the process and metric should be able to do), specification (of processes and constraints), testing (do the process and OCL constraints really enforce the intent of the process or QA engineer), and reuse (how to efficiently reuse processes and constraints across teams and tools).

We also need to understand how process guidance might negatively affect engineering activities. In preliminary experiments with students, we noticed that participants were tempted to conduct quicker, but semantically inappropriate fixes to fulfill a violated constraint if they didn’t manage to execute a semantically suitable fix just to obtain a green check mark on the guidance front end. More research is needed to understand how such behavior can be avoided.

We also plan on improving the guidance capabilities in terms of support for temporal constraints (up to now, they were not in demand by our industry evaluation partners) and the guidance actions for violated temporal constraints. Envisioned extended capabilities also address the challenge of identifying not just a plausible guidance action but rather ranking these actions to provide the most suitable one for a particular engineering situation at the top.

Finally, our experience has shown that focusing only on existing engineering meta informa-
tion and traces among artifacts creates a too narrow view of process guidance. Analyzing the content of engineering artifacts in more detail is expected to allow a more precise assessment of the engineering progress, e.g., understanding to what extent high-level requirements have been refined low-level requirements and subsequently being able to assess when engineering tasks are truly complete.
5 Publications

Following publications are part of this habilitation thesis:


5.1 Supporting Quality Assurance with Automated Process-Centric Quality Constraints Checking

Authors:
Christoph Mayr-Dorn, Michael Vierhauser, Stefan Bichler, Felix Keplinger, Jane Cleland-Huang, Alexander Egyed, Thomas Mehofer

Venue:
43rd IEEE/ACM International Conference on Software Engineering ICSE (2021)

DOI:
https://doi.org/10.1109/ICSE43902.2021.00118

Abstract:
Regulations, standards, and guidelines for safety-critical systems stipulate stringent traceability but do not prescribe the corresponding, detailed software engineering process. Given the industrial practice of using only semi-formal notations to describe engineering processes, processes are rarely “executable” and developers have to spend significant manual effort in ensuring that they follow the steps mandated by quality assurance. The size and complexity of systems and regulations makes manual, timely feedback from Quality Assurance (QA) engineers infeasible. In this paper we propose a novel framework for tracking processes in the background, automatically checking QA constraints depending on process progress, and informing the developer of unfulfilled QA constraints. We evaluate our approach by applying it to two different case studies; one open source community system and a safety-critical system in the air-traffic control domain. Results from the analysis show that trace links are often corrected or completed after the fact and thus timely and automated constraint checking support has significant potential on reducing rework.
Supporting Quality Assurance with Automated Process-Centric Quality Constraints Checking

Christoph Mayr-Dom
Johannes Kepler University
Linz, Austria
christoph.mayr-dom@jku.at

Michael Vierhauser
Johannes Kepler University
Linz, Austria
michael.vierhauser@jku.at

Stefan Bichler
Johannes Kepler University
Linz, Austria

Felix Keplinger
Johannes Kepler University
Linz, Austria

Jane Cleland-Huang
University of Notre Dame
Notre Dame, USA
JaneHuang@nd.edu

Alexander Egyed
Johannes Kepler University
Linz, Austria
alexander.egyed@jku.at

Thomas Mehofer
Frequentis AG
Vienna, Austria
thomas.mehofer@frequentis.com

Abstract—Regulations, standards, and guidelines for safety-critical systems stipulate stringent traceability but do not prescribe the corresponding, detailed software engineering process. Given the industrial practice of using only semi-formal notions to describe engineering processes, processes are rarely "executable" and developers have to spend significant manual effort in ensuring that they follow the steps mandated by quality assurance. The size and complexity of systems and regulations makes manual, timely feedback from Quality Assurance (QA) engineers infeasible. In this paper we propose a novel framework for tracking processes in the background, automatically checking QA constraints depending on process progress, and informing the developer of unfilled QA constraints. We evaluate our approach by applying it to two different case studies; one open source community system and a safety-critical system in the air traffic control domain. Results from the analysis show that trace links are often corrected or completed after the fact and thus timely and automated constraint checking support has significant potential on reducing rework.

I. INTRODUCTION

Software quality assurance (QA) focuses on ensuring and attesting that the engineering processes result in appropriate quality of the software. To this end, various regulations, standards, and guidelines stipulate stringent traceability paths [1], [2], but don't prescribe the corresponding, detailed software engineering process. Examples in safety-critical systems include the FDA principles in the medical domain, DO-178C/ED-12C for airborne systems, ED-109A for air traffic management systems, and Automotive SPICE in the automotive industry. Here, QA engineers need to inspect fine-grained constraints over properties of engineering artifacts (i.e., requirements, models, code, test cases, etc.) as well as trace links at specific points in time (i.e., in different process steps such as requirement elicitation, specification refinement, coding, or test case specification). The current practice in industry, however, is to use semi-formal descriptions to specify processes [3]. As a result, software engineering processes are rarely "executable", meaning that there is little to no automated support for checking whether these processes are followed, and to what extent deviations occur during development.

In this paper we specifically address the problem of Developers and Quality Assurance Engineers being overwhelmed by the complexity and extent of adhering to, and evaluating, QA constraints. Typically, developers work on multiple projects, sometimes simultaneously, with each project having different quality standards or guidelines. In an informal study with our industry partners, developers reported being stressed about potentially missing important steps mandated by quality assurance. QA Engineers, on the other hand, need to conduct countless, tedious, often mind numbing checks that involve (manually) navigating across diverse artifacts and tools to ensure that the required constraints are fulfilled at the right process step. These checks are error prone and rarely conducted in time to provide immediate feedback to developers. Remediating problems later on, however, interrupts developers who may have already moved on to other steps or projects, causing even further delay.

In this paper we present a novel approach that aids developers and quality assurance engineers to reduce the effort required in ensuring that development activities adhere to the intended process - ultimately leading to less rework. Our approach relies on passive process execution, achieved by tracking process steps via monitoring engineering artifacts such as requirements, design documents, issues, change requests, and tests. This is complemented by constant evaluation of quality constraints. The key novelty is treating (quality) constraints neither as an implicit part of the engineering process model nor as completely disjoint from it. Instead, we propose treating (quality) constraint evaluations as first class citizens: i.e., as explicit development artifacts that determine process progress. This contrasts with existing work on traceability [2], [4] where links required by regulations and standards are typically verified by an auditor at the end of a process stage or prior to shipping the final product [5]. Similarly, work on constraint checking [6], [7] primarily focuses on consistency among diverse artifacts without addressing consistency issues.
between these artifacts and the underlying process.

The key contributions of this paper are as follows:

- A process model that decouples process control and data flow from QA constraints.
- A passive process engine which explicitly tolerates inconsistencies \[8\] and allows engineers to temporarily deviate from the process while providing them with timely feedback on QA constraint evaluation results.
- A prototype that helps developers clearly understand when they have completed a step or what they still need to provide (e.g., specific content or trace links).
- An evaluation against an open source system for unmanned aerial vehicles (UAVs) and an industrial air traffic control system (ATC), that measures the extent to which the prescribed process was followed.

The remainder of this paper is structured as follows. Section II motivates our work by describing constraint checks for the development of safety-critical systems. Sections III and IV provide an overview of our approach and introduce details on modeling the process and constraints. We then evaluate our approach using two distinct case studies in Section V, and conclude with a discussion of results (Section VI), threats to validity (Section VII), and related work (Section VIII).

II. Motivating Example

Our industry partner Frequentis is a world leading voice communication provider for air-traffic control and command-control centers. In this domain the DO-278/ED-109 standard \[9\] specifies traceability requirements according to different design assurance levels, ranging from “Catastrophic” to “Minor” or “no Effect”.

However, while DO-278/ED-109 defines which trace links need to be available, it does not specify when these trace links need to be established or who should/must perform this task.

Figure 1 (left) depicts a partial traceability information model (TIM) and excerpts from a simplified development process representation (right) for one Frequentis product. The process example indicates that once high-level requirements (HLReq) and high-level design specifications (HLSpec) are reviewed, HLReq may be refined into low-level requirements (LLReq) and cause updates to the low-level design specification (LLSpec). Thus, developers should not start refining LLReq without the outcome of the review, even though HLReqs and teams are already assigned to work packages during the development iteration planning phase. Additionally, before the implementation of LLReq can start, trace links from HLReq to LLReq and trace links from LLReq to LLSpec need to be reviewed. Constraints that are set up for the development process are not only used to check for the existence of an artifact or the existence of a trace link between the right type of artifacts. Constraints build on the properties of an artifact at a specific process progress. At the end of step S3 (cf. Figure 1) the respective LLReqs need to be in state “Complete”, each having a trace link to a HLReq. Upon completion of step S7 when a LLReq’s verification method is “Demonstration”, then there must exist at least one trace link to a respective Test Case which is not of type “Software Test Case” (but, for example, a simulation, demonstration, or acceptance test). These constraints are complementary to human-in-the-loop QA measures such as the trace reviews in S5 and S6. The automated checks reduce the effort for these reviews by ensuring the traces under review are syntactically correct and thus the review can be done more efficiently.

Engineers who are updating the LLSpec (in S4), for example, thus need to be aware when they may start with their step to avoid rework if additional refinements of an LLReq occur. Similarly, engineers in S7 need timely feedback when they can claim to have finished implementation and thus trigger the review in S8.

Knowing the state of the process helps assess the risk of deviation. Starting early on HLReq refinement (S3) may be too risky if none of the requirements have gone through review in S1 and S2 but perhaps necessary and ok due to time pressure when S2 has only a few requirements left to review. The following sections use this TIM, process, and constraints as a running example to better explain process and constraint modeling, as well as execution.

III. Approach – The ProCon Framework

In this section we provide a comprehensive overview of ProCon, our passive Process execution and quality Constraint support framework. Two key aspects that characterize our framework are: first, the integrated handling of explicitly distinct processes and constraints, and second, the tracking of engineering progress achieved through explicitly linking process descriptions to software engineering artifacts.

In ProCon, a passively executable Process Specification describes the sequence and alternatives of carrying out engineering activities (i.e., the “control flow”) and the software engineering artifacts that serve as inputs and outputs of those activities (i.e., the “data flow”). Explicitly modeling a software engineering process for controlling the software engineering life-cycle is not new, with a plethora of research dating back until the 90s \[10\]-\[13\]. ProCon is different insofar as that (i) the process is tracked in the background, based on the

![Traceability Information Model (TIM) excerpt (left) and process model excerpt (right).](image-url)
software engineers’ activities performed in the tools they are using in their daily work rather than requiring engineers to interact with a process engine, (ii) engineers are free to deviate from the process, (iii) engineers may receive guidance even in the presence of deviation, and (iv) ProCon supports control-and-dataflow conditions as well as constraints across diverse artifact types and tools.

We refer to these abilities as “passive execution” as our framework goes beyond simple process monitoring by determining available future steps, detecting premature steps, and making this information immediately available to engineers.

With ProCon, deviations from the process are tracked via process and quality constraints. These constraints, and their respective evaluation results, are treated as first-class citizens for explicitly evaluating constraints as soon as actions are made. Constrains and their evaluation results as first-class citizens make reuse and maintenance less cumbersome. Constraints such as issues often serve as a (partial) informal representation of process instances. Tool Connectors (B) for the respective tools provide (machine readable – e.g., as JSON data via a REST API) access to artifacts. These connectors take on sophisticated tasks such as obtaining artifact updates (via polling or subscriptions), managing which artifacts are relevant for a process and thus need to be monitored, caching the artifacts for quick repeated access, and keeping this cache up to date (cf. Section V-A).

Passively Executable Process Specifications and Constraints (C) are manually derived (cf. Section IV-A and IV-B, respectively) based on the above two main inputs. Process Specifications formalize the (semi-)informal process definitions and guidelines and allow a fine-grained mapping to the respective artifacts, properties, and their changes observable via the tool connectors.

The Passive Process Engine (D) manages instances of passively executable process specifications (or Process Instances for sake of brevity). The passive process engine obtains artifact state and events from tool connectors (E), feeds these changes into a rule engine in which the process progress conditions and QA constraints are evaluated. The rule engine eventually fires events that the process engine utilizes to update the process progress and quality constraint evaluation results (cf. Sections IV-C and IV-D, respectively).

A web-based Process Dashboard makes Process progress and QA constraint evaluation results (F) continuously available to software engineers. It allows controlling the passive process instance: triggering new instances, observing their progress, and eventually archiving them (cf. Section V-A).

B. ProCon Usage

ProCon users are primarily QA engineers and developers, but also include stakeholders such as product managers and team leads. The former can use the framework with a focus on quality assurance (the focus of this paper), while the latter can use the framework with a focus on process progress.

Based on this, ProCon supports two distinct use cases. In the process and constraint modeling use case, various stakeholders map the informal process definitions, standards, etc. (A) to passively executable process specifications (C). This phase involves analyzing how engineers currently produce evidence of process execution in the various tools (B). The tool connectors describe what artifact properties are available and thus what change events may serve as process progress triggers during process execution: for example, what (custom) properties are used for various Jira issues, what trace links are available to navigate from a Jira [16] Issue to a requirement.
managed in Jama [17]. The outcome of the modeling use case is a passively executable process specification and its quality constraints.

Note, that our approach does not require modeling the complete process if tools don’t allow access to certain information. One would typically start with those parts that are most important, or most error prone, etc and focus on these. Ambiguity may arise, for example, if rules expect one trace to navigate to an artifact but find multiple ones, then a random one will be selected. Mitigation includes correcting and/or extending QA-constraints to identify the ambiguous situation.

In line with Lee Osterweil’s key observation that “software processes are software, too” [18] we advocate the use of contemporary software engineering practices such as developing in iterations, testing, versioning, and issue tracking. A key element here is replaying of the artifacts’ history to test the constraints’ ability to detect deviations [19].

The second use case focuses on the passive process execution. Upon process instantiation, the passive process engine obtains artifacts and their changes as they occur in tools (D). The engine then tracks the progress of each step and attached constraints. It determines completed and in progress steps (Figure 2 center, with solid border), which steps an engineer is free to start next, and which ones should not yet start (dashed border). Artifact and process updates trigger reevaluation of constraints (document symbols with icons). ProCon users then access the process progress and constraint evaluation results (E). An engineer may notice an unfulfilled constraint, conducts the necessary artifact changes via the tools, triggers reevaluation, and confirms quality constraint fulfillment. Note that users exclusively affect process progress and constraint evaluation results via tool interactions (F) and never via direct interaction with the passive process engine itself (except for explicit triggering of quality constraint evaluation). In the next section we describe the process and constraint modeling in more detail as well as process execution.

IV. PROCESS, CONSTRAINT MODELING, AND EXECUTION

The following section describes how Process Specifications are structured and executed, and discusses the elements and transitions of the ProCon Process Specification metamodel.

A. Process Specification Model

The challenge when passively executing processes is determining the steps that are currently available for engineers to work on, steps that represent work in progress (but perhaps shouldn’t be worked on yet), and finally, steps that have been (successfully) completed. The prevalent differences to existing approaches hereby are not the basic building blocks (i.e., the process steps) but rather in how transitions between these steps are defined and subsequently triggered. Figure 3 (left) provides a simplified UML class diagram of the process model’s main elements. The two main elements of the process model are Steps and Decision Nodes attached to them.

A Step describes what an engineer “should” do (in contrast to “must” do – as prescribed by a a more restrictive traditional process). For example, refine a requirement, implement a feature, define a test case. A step has zero or more Input artifacts attached that represent required data to make a decision or artifacts that need to be modified. It further has zero or more Output artifacts that describe the effect of having executed the step (e.g., having modified an input artifact or created a new artifact). Input and output artifacts can represent any kind of information such as requirements, tests, issues, or trace links. In addition to the textual description of an engineer’s activity, a step consists of a set of event-condition-action rules that define which event(s) from the engineering environment (e.g., an artifact update), given additional constraints (i.e., the condition) trigger the inclusion of an artifact to a step’s output artifact set (i.e., the action part of the rule). For example in S5, “When an engineer posts a review URL as a Jira issue comment, then add that link as the step’s output artifact”.

A Decision Node describes how the completion of one or more Steps – and additional conditions – leads to the execution of subsequent steps. The set of decision nodes thus defines the process’ control flow. A decision node’s DataTransfer declaration describes how the output of one step becomes the input of a subsequent step, thereby defining the process’ data flow. For example, “The LLReq output artifacts of S3 and LLSpec output artifacts of step S4 become the input artifacts to step S6”. Note that a step may only have one preceding and one subsequent decision node to avoid conflicting control or data flow. Only a decision node may link to multiple steps. Ultimately, a process consists of a set of steps and decision nodes that create a single connected, directed graph. Note that currently loops are not yet supported as the use cases at our industry partner did not require this feature for two reasons: first, artifacts can be updated over and over again until QA-constraints are fulfilled. Second, longer, explicit loops such as sprints are typically represented as separate sub/processes and thus are “spawned” separately.

The specific activation conditions are placed on the InFlows (from preceding steps to decision node), on the node itself, and on the OutFlows (from decision node to subsequent steps). Figure 3 (right) outlines how these conditions reflect in a decision node’s state.

- AVAILABLE: upon instantiation, a decision node is in the AVAILABLE state. Depending on a decision node’s InFlowType: AND, OR, XOR, it will check upon each update of preceding steps whether all, at least one, or exactly one InFlow is fulfilled to transition the node into PASSED_INCONDITIONS.
- PASSED_INCONDITIONS: in this state, conditions independent on any preceding step, such as process or date-time centric conditions need to be fulfilled before transitioning into PASSED_CONTEXTCONDITIONS.
- PASSED_CONTEXTCONDITIONS: in this state, the decision node evaluates OutFlows. OutFlow conditions serve as a filter to dynamically select the relevant subsequent steps to enable. Depending on a decision node’s OutFlowType (AND, OR, XOR), it will check whether all or at least one OutFlow is fulfilled, then transitioning...
As long as a decision node hasn’t reach PROGRESSED, any change to the InFlow, context, and OutFlow conditions will transition the state back to an earlier state.

Note that the conditions on InFlow, context, and OutFlow are optional. For example, the decision node between step S1 and step S2 (not shown in Figure 1) does not have any of these three conditions. Upon completion of S1 it will immediately transition through all these states into state PROGRESSED, thus serving as a simple trigger to activate step S2.

In addition to the decision node state, tracking each step’s state is vital to provide developers with feedback on which steps are ready for starting, which ones have started too early, and which ones should not be done. Figure 3 (middle) depicts a step’s life-cycle modeled as a finite state machine.

- AVAILABLE: when a step is instantiated it resides in the AVAILABLE state, indicating that it’s input is not sufficient yet (i.e., it has not yet obtained the necessary data from the preceding decision node).
- ENABLED: once all specified input conditions are met (for example, required input artifacts are available), a decision node causes the step to transition to ENABLED, indicating that an engineer is free to start working on it.
- ACTIVE: when Artifact-to-Step Mappings signal that artifacts attached to the state are updated or modified, the step becomes ACTIVE, indicating that an engineer is actively working on it.
- NO WORK EXPECTED: when multiple mutual exclusive steps are ENABLED it can initially not be determined which of these an engineer has chosen. Once, one of the alternative steps transitions into ACTIVE, the remaining steps transition into NO WORK EXPECTED.
- REVOKED: when a step’s input conditions are no longer fulfilled, it transitions into REVOKED to indicate that an engineer should not/no longer work on this step, or if starting, respectively continuing, may need rework later on. Once the input conditions are re-established, the step transitions back into ENABLED or ACTIVE (depending on its previous state). The key difference between a REVOKED step and NO WORK EXPECTED step is that the former should eventually be carried out, while the latter should not be carried out at all.
- COMPLETED: when all output conditions are met (e.g., all required output artifacts are available), the step transitions into COMPLETED and triggers the evaluation of the step’s subsequent decision node.

B. Quality Constraint Integration

For each step, Quality Constraints can be defined describing conditions the created or updated output artifacts must adhere to. A constraint can refer to a step’s input and output artifacts, its metadata, or process metadata.

Each constraint has an identifier that allows triggering constraints not only upon artifact changes but also manually, on demand. This is important to provide control to the user and to allow, for example, an engineer to trigger a constraint check to reassure him/her that a step is indeed complete and nothing has been overlooked. The language in which constraints are implemented is flexible and can vary depending on the domain or application scenario. The only requirement is that it has to provide respective evaluation results (we provide further examples on constraints and the constraint language as part of the use cases in Section V).

Every time a quality constraint is evaluated, a corresponding new instance of Quality Constraint Evaluation Result is created, reporting the result (i.e., fulfilled or unfulfilled) and lists the artifacts subject to that constraint. For example, a constraint for step S6 checks whether each low-level requirement (LL-Req) output artifact has a trace link to a high-level requirement (Hl-Req) which must be in state “released”. Therefore, the respective constraint evaluation result will contain the list of LL-Req s that fulfill this constraint, and a list of LL-Req s...
that violate this constraint. For every constraint, we further store timestamps indicating when the constraint evaluation result was (last) evaluated, and when the evaluation result last changed from violated to fulfilled and vice versa. This, for example, allows to infer how recent results are when shown on the Process Dashboard. The evaluation results of all quality constraints associated with the same step are collected and bundled in a Quality Check Document which is added to the output artifacts of that step.

The process model itself remains largely independent from constraints and their evaluation results. While a process step’s completion constraint must check whether a Quality Check Document output artifact exists which contains only positive Quality Constraint Evaluation Results, it doesn’t need to understand the particular constraints that resulted in the evaluation success. This loose coupling allows to execute the same process with different levels of quality assurance by switching in and out different quality constraints.

C. Passive Process Execution

Every process model comes with an activation condition, typically an explicitly added artifact such as a change request issue or work package issue that already serves as some form of process representation. The process engine instantiates the process and checks which steps and decision nodes can be instantiated based on the provided activation artifact. Instantiation occurs incrementally, i.e., only when a step reaches the state ENABLED, will the process engine instantiate the step’s subsequent decision node. Similar, only when a decision node is in state PASSED_OUTFLOW_CONDITIONS, will the engine instantiate the steps identified by the outflow links. With step instantiation, the engine also instantiates the quality check constraints. Thus, as long as a step doesn’t exist, none of its constraints will be checked.

In the passive process engine, incremental step and decision node instantiation is insufficient as engineers may decide to work on steps not yet ready. When artifact changes trigger Artifact-to-Step Mappings, the process engine will instantiate the corresponding “premature” step and attempts to find an existing preceding decision node that this step can be linked to. If none exists yet, the step remains dangling in the process until the process progress catches up; i.e., upon instantiating a decision node, the engine checks if a dangling step exists that should be linked via one of the decision node’s out flow links. From the engine’s point of view, there is no difference between missing a step and starting the next, or starting too early on the next step. It will continue either way. In such a case, however, the engine will not be able to fully execute a DataTransfer that requires the output of the skipped/incomplete step. The consequence is then highlighted via the step’s status as having insufficient input artifacts.

Note that assessing the impact of prematurely starting a step and how to mitigate any potential change propagation is outside the scope of this paper. Premature steps stand out from regular steps by having transitioned directly from AVAILABLE to ACTIVE.

D. Constraint Checking

We made the deliberate decision to execute constraint checks upon explicit request and not automatically triggered by every single artifact change. Often a single change is not indicative of step completion. The reasons behind this are manifold. First, as quality constraints often span across multiple artifacts, a single change to an artifact is insufficient, multiple changes need to occur. For example in S6: not just one low level requirement (LLReq) needs to be set to “released” but all linked ones. Second, an artifact may be involved in multiple constraints, thus a change would trigger execution of multiple constraints. Third, reevaluation on every change puts unnecessary burden on the process engine, respectively its constraint evaluation subsystem. Forth, when quality constraints involve diverse artifact types managed in different tools, change events from these tools may not be readily available as the engine carries out the change in these tools but need to be fetched periodically. Jama, for example, doesn’t offer automatic event notifications but requires polling with subsequent explicit fetching of changed artifacts. This would result in executing checks on stale data.

V. EVALUATION

We evaluated ProCon against two distinct use cases for which we created process specifications and constraints, and implemented connectors for different issue tracking and requirements management tools. The first use case, Droolology [20] – an open source project – represents an more agile, lightweight process, whereas the second one – a safety-critical system in the air traffic management domain – describes a rigid, standardized process with stringent quality assurance criteria. We report on the application of ProCon to the two use cases and our findings and lessons learnt. Supplemental online material [21] provides artifact data and trace links, process definitions including constraints, and the experiment results. The prototype source code is publicly available [22].

A. Prototype Implementation

We implemented a prototype to evaluate the two use cases and to obtain feedback from our industry partner Frequentis.

Tool Connectors: To cover a reasonably large set of artifacts from our industry partner Frequentis and from Droolology, we implemented connectors for Jira, a web-based tool for planning, issue tracking, and reporting, and Jama, a tool for requirements management, traceability, and test management.

The Jira Connector uses the Atlassian Java REST API to retrieve artifacts and their attributes and is used to periodically poll for changes in these artifacts. Similarly, the Jama connector uses the Jama REST API. To reduce load on network and tools, the tool connectors cache Jira and Jama artefacts in a CouchDB (a schemaless JSON database).

Process Engine: The Process Engine is implemented in Java, containing an implementation of the process specification metamodel and a rule engine for checking constraints. We opted for the Drools rule engine [23], a Business Rules Management System that can be easily integrated into a Java
application and allows easy access to Java objects (representations of Jira and Jama artifacts) within rules written in a Java dialect. Additionally, we persist the processes and Quality Check Documents (including Quality Constraint Evaluation Results) attached to the different process steps in a Neo4J graph database. The graph structure simplifies fetching all related process steps, decision nodes, input and output artifact (references), and quality check documents in a single query to be shown on the process dashboard.

**Passive Process Specifications and QA Constraints:** The Drools rule engine evaluates process progress conditions and quality constraints. We therefore defined quality constraints as well as the decision nodes' control and dataflow conditions in respective Drools rules files.

**Process Dashboard:** Figure 4 shows the user interface for inspecting quality constraint evaluation results. The results contain browseable links to the original artifacts thus enable the engineers to quickly switch into their usual tools (here Jama and Jira) to fix any unfilled constraints.

B. **Open Source Use Case: Dronology**

Dronology is a UAV management and control system providing a full project environment for managing, monitoring, and coordinating the flights of multiple UAVs. It can interact with real hardware as well as a high-fidelity Software-in-the-Loop simulator that enables experimentation with virtual UAVs. Dronology was developed, with both students and professional developers, over several years as a research incubator with various development artifacts publicly available [20], [24]. For the purpose of this case study we obtained permission to use data from multiple sprints maintained in Jira from 2017 to 2019. This includes the following artifacts: Bugs, Hazards, Requirements, Design Definitions, Tasks, and Sub-Tasks.

**Process and Constraint Creation:** We treat each of these issues as “small sub processes” where the issue’s state represents a process step, and quality constraints for each step describe the conditions that need to be fulfilled to transition from one step to the next, respectively complete the process (i.e., close the issue). Given the lean nature of typical agile open source development processes the states were limited to the default process steps in Jira, “Open”, “InProgress”, and “Closed”. Based on the information available, we identified the following eight quality constraints and allocated them to the steps where they are most useful (note that some constraints are reusable for multiple issue types). Process and constraint creation took around 3 hours. We then contacted one of the lead developers to confirm the validity of the constraints and the process. An overview of the constraints can be found in Table I. At the end of step “Open”, we require constraints D-C1 to D-C5 to be fulfilled, and at the end of step “InProgress” we require constraints D-C6 and D-C7 to be fulfilled.

C. **Industry Use Case: Frequentis**

For the Frequentis use case we selected a safety-critical product for voice communication in air traffic control centers. The product consists of several subsystems for interfacing with radio transceivers, managing near-real time voice streams, and providing operator user interfaces. Frequentis follows a V-model like engineering process. Specifically, for this case study, we focused on sub work packages (SubWP). This process for each team (each responsible for one subsystem) starts with high-level requirements resulting in the actual implementation and the successful execution of test cases. This covers steps S3 to S8 of the motivating scenario. Trace links between SubWPs and low-level requirements are therefore the main focus of QA constraints defined as the completion conditions of steps S3 (represented by ATC-C1 to ATC-C4), S4 (ATC-C5), S7 (ATC-C6 and ATC-C7), and S8 (ATC-C8) (cf. Table I). Frequentis uses Jama to manage all artifacts and trace links depicted in Figure 1 and uses Jira to manage the engineering process.

**Process and Constraint Creation:** We defined the constraints together with a QA engineer within two hours, and took another two hours to specify the process. Frequentis’ informal process definition precisely defines how engineers need to set properties of Jira and Jama artifacts for completing the various steps. Changes to these properties serve as step completion signals in our process engine.

D. **Data Gathering**

We were granted access to the Dronology project’s Jira server REST API to obtain artifacts and their change history. The data set consists of 802 process instances (i.e., Jira issues): 199 Tasks, 211 Sub-tasks, 109 Bugs, 247 Design Definitions, and 36 Hazards. From Frequentis we obtained Jira issues related to the aforementioned SubWPs. Each SubWP managed in Jira has a corresponding Jama artifact with respective trace links to LLReqs and subsequent artifacts. We used the Jama REST API to navigate across these trace links to collect all Jama artifacts (including their history) that are relevant for constraint evaluation. This resulted in a set of 109 SubWPs and ~14,000 linked Jama items (out of which 1,121 are LLReqs).

We used our trace link replay tool [19] to reset the entire state of the dataset of Jira issues and Jama items and their trace links to the earliest change event and then replayed...
Table II reports the QA constraint evaluation results across multiple process instances, grouped per process type. Row *AlwaysOk* shows the numbers of process instances where engineers only progressed to subsequent steps when all quality constraint in previous steps were fulfilled. Row *EventualOk* shows the processes for which all constraints were eventually fulfilled. The row *CompleteNotOk* shows processes for which some constraint were never fulfilled. Row *IncompleteOk* counts those process instances that were not finished by the end date of the timeframe but had all mandated constraints up to their current state fulfilled. Row *IncompleteNotYetOk* counts the partially completed process instances with unfulfilled constraints but no progress beyond those not fulfilled steps, in contrast to those with progress beyond in row *IncompleteProgressedNotOk*. Percentage values are reported relative to the sum of completed process instances, respectively sum of incomplete process instances.

**Dronogy:** we noticed that for “Task” processes no completed process instance (i.e., finished “Task”) ever fulfilled every constraint before moving from one step to the next, yet around 30% fulfilled all their constraints at the end, with ~70% remaining unfulfilled at the end. “Sub-task” processes see ~30% of instances “correctly” carried out, with only ~50% not fulfilling their constraints. “Bug” processes are almost always correctly executed. “Design Definition” and “Hazard” processes are either correctly carried out from the beginning (the vast majority), or remain with unfulfilled constraints. Observing the incomplete process instances we encountered an expected large number of processes with unfulfilled constraints (i.e., hinting at steps with associated QA constraints that are not complete yet). However, we notice that only in a low percentage (< 20% for *IncompleteProgressedNotOk*) of instances have engineers started too early on subsequent steps without having fulfilled the previous steps’ constraints.

**Frequentis:** we noticed two aspects. First, the amount of SubWPs ultimately Ok reaches almost 90%, with 10% SubWPs remaining with unfulfilled constraints. We manually investigated the violating artifacts (exclusively LLReqs) and the comments attached to the SubWP Jira issue. Given that Jira is used as the primary communication and coordination mechanism amongst the distributed teams and QA department, the comments provide an accurate and sufficiently complete track of the SubWPs history. For the 10 CompleteNotOk SubWPs we found that in two cases SubWPs were used for documentation purposes rather than development (thus no trace links to Functional Units were present). In one case test cases were not applicable, and in three cases multiple Functional Units were linked rather than one. This was due to the fact that the configuration subsystem affects multiple Functional Units. Three times a test case was referenced in the Jira comments (but no corresponding trace link in Jama was created). Once an additional SubWP was traced without closing the older one, and three times LLReq were marked for proposed future changes (and thus being no longer in state “released”). Note that some SubWPs experienced multiple, diverse violations. Second, we found that 11 SubWPs are IncompleteOk even we know that all the work was done. Manual investigation revealed that the Jira custom fields which are used by the passive process engine as a signal to advance every single change in the correct temporal order. The changes occurred between April 2017 and December 2019 for Dronogy and between May 2018 and June 2020 for Frequentis, respectively. Using the replay tool allowed us to start from the beginning of the development process and, step-by-step, simulate (i.e., "replay") changes made by engineers (e.g., modify the state of artifacts in Jira, add trace links, etc.) allowing us to automatically trigger constraint checks and track the process state the same way as in a "live" environment. For each constraint evaluation we evaluated (i) whether a step’s Quality Check Document was fulfilled; (ii) which constraints were (not) fulfilled; and (iii) whether a step became active without the constraints of the previous one(s) being fulfilled.

Additionally, for a process instance (i.e., a Jira artifact), we collected the following metrics: number of Quality Check Documents un/fulfilled; number of un/fulfilled constraints; number of constraint checks performed; and the maximum number of past steps with unfulfilled constraints (i.e., how many steps an engineer advanced ahead without having the completion condition of the previous steps fulfilled).

### E. Process Replay Results

Table II shows the QA constraint evaluation results across multiple process instances, grouped per process type. Row *AlwaysOk* shows the numbers of process instances where...
Constraint Fulfillment - Dronology: Table III reports the differences in how often a constraint was fulfilled (limited to completed process instances). A majority of constraints was fulfilled most of the time (~90% and higher). The lower fulfillment rates for constraints D-C1a and D-C1b (<55%) are the main reason “Task” and “Sub-task” processes exhibit low AlwaysOk and EventualOk values in Table II. Yet, constraints applied across multiple process types (i.e., D-C1a/b, D-C2, D-C3, D-C6, D-C7) exhibit similar fulfillment rates.

Frequentis: For this case we generally observed high fulfillment rates for all constraints. The 12 unfulfilled constraint instances are distributed across the 10 CompleteNotOk SubWPs described above. Recall, a Dronology constraint typically requires the existence of a trace link to one artifact (e.g., D-C1a: a Task traces to a least one Design Definition). In contrast, for Frequentis a constraint requires that all linked artifacts (i.e., LLReqs in ATC-C1 to ATC-C6) fulfill specific conditions. For ATC-C5, for example, a single LLReq out of 10 that doesn’t have a trace link to a Functional Unit will cause the entire constraint to fail (regardless of whether all other LLReqs are correct). We, therefore, also looked at the number of times an artifact (primarily an LLReq) was part of a constraint violation. With 1,121 LLReqs and six constraints involving an LLReq, there are potentially 6,726 opportunities that cause an overall constraint to fail. We observed 128, which is less than 2%. Out of 128 LLReqs that were part of a violation (due to missing, wrong, or too many trace links) only 3 of these cases were part of two different constraint violations. 98 LLReqs belonged to a single SubWP that was used for documentation (and needed no Functional Unit trace links), additional 12 LLReq belonged to single SubWP where Test cases were not applicable. The remaining 18 LLReqs violations were spread across the other eight CompleteNotOk SubWPs.

F. Performance

An engineer typically checks constraints at the level of Quality Check Documents to ensure all QA demands for their step are fulfilled (and not necessarily for each individual constraint). Overall the replay of 26,926 change events over 109 simultaneously active process instances resulted in 18,241 Quality Check Document evaluations. The resulting replay of events from the Tool connector’s cache including constraint evaluation took ~6.5 minutes (averaged over 10 evaluation runs). This corresponds to ~0.02 seconds necessary for evaluating all quality constraints within a single Quality Check Document: a duration that allows frequent and timely feedback to developers.

VI. Discussion

The analysis of the Dronology data set indicated that the actual process (in some cases significantly) deviated from the planned process. Upon requesting feedback, a project lead at Dronology explained that while guidelines and a development process were in place, it was not always feasible to follow them by the letter. Student teams were involved in the development of some of the components, and while they have been trained on the process, they still lacked experience in following all prescribed rules and guidelines. Furthermore, besides the software development aspect, the focus was also on obtaining a data set of trace links and that the process had to be adapted to the availability of open source developers. Rather than forcing a change of process which might be infeasible, the insights gained here could be used to decide where to better place the QA checks, e.g., making constraint check results available upon reviewing a pull request. Here ProCon would then highlight where traces are missing or are incorrectly set. A trace recommendation technique such as [25], [26] could further assist in establishing the trace itself.

In contrast, the analysis for Frequentis’ SubWPs confirmed that engineers followed the high and stringent quality standards expected in the (highly safety-critical) ATC domain. The finding that 10% of process instances “EventualOK” confirms the QA engineer’s experience that engineers need support for producing correct and complete trace links as significant additional work at a later stage was necessary. Our investigations into the “CompleteNotOK” instances highlighted that, on the one hand, corrections come with significant coordination effort and still may result in missing traces or incorrectly set artifact properties. On the other hand, the investigations highlighted the presence of edge cases where the QA constraints do not apply, reinforcing the need for sometimes tolerating these inconsistencies. ProCon offers two options in such a case: first to ignore the constraint evaluation results, and/or to adapt the process, respectively constraints. In either of these two cases, a rigid (thus inflexible), active process enforcement environment would have severely hampered the engineer’s available actions, effectively forcing the engineer to work outside the defined process. Finally, the huge amount of >18,000 Quality Check Document evaluations explains why manually providing timely feedback is infeasible.

Using ProCon can have significant practical implications for QA engineers. Supported by automated checks for “standard” cases, they can shift attention and focus their time on edge cases and deviations from the process. Furthermore, they can allocate time for improving constraints checks, and investigating whether these checks and following the process actually result in better software quality [27]. Engineers can leverage the immediate feedback they receive on their work status and do not need to revisit their work at a later, inconvenient time. The various stakeholders no longer need to build (error prone) custom “helper tools” that are hard to maintain or to reuse.

We received very positive responses from engineers at Frequentis upon presenting ProCon with one team lead wishing to have it ready as a product by tomorrow, and a QA engineer joking to be out of work then. QA engineers at Frequentis used the prototype for writing new QA constraints during a company internal innovation event to showcase its potential and adaptability. While the prototype was applied only to one
product group at Frequentis, we are currently rolling out the prototype to three more product groups, each having different rules (but use Jira/Jama), thus only the process and rules needs to be adapted. Given the excellent performance during replay (i.e., handling 27k artifact changes across 109 process instances within a few minutes) we are confident that adding more rules in the current rollout will not lead to performance problems. We subsequently expect to obtain more detailed insights into the prototype's practical use.

Aside from immediate practical implications, ProCon has huge potential as the foundation for additional support tools building on top. Passive process execution has the benefit of enabling inspection at any time to what degree the process is followed and where deviations have occurred (respectively are not mitigated yet). Deviations can thus be detected earlier, e.g., an engineer has started too early on a step. Alerts or mitigating actions may then be less invasive rather than significant rework later on. Other potential support mechanisms could guild the engineer in how to setup the correct output artifacts, or direct the engineer in how to fix a constraint violation or offers to automatically fix it [28].

VII. THREATS TO VALIDITY

Internal Validity. We address researcher bias by modelling process and constraints from an open source system and a company rather than conducting controlled experiments. ProCon works on arbitrary artifacts, traces, and change events and was not specifically tailored to Jama or Jira.

External Validity. Based on the limited scope of our evaluation with two different systems, we can not claim generalizability of our findings. However, we argue in line with Briand et al. [29] that context-driven research will yield more realistic results. Our work evaluated the ability of ProCon to passively execute diverse engineering processes and QA constraints (simple ones from an open source system and more complex ones from industry) in a timely manner. We analyzed data from these two sources with one being “production data” from an industrial safety-critical system. Typically, being able to obtain such data, and furthermore being able to publicly report results is quite challenging as companies are reluctant to provide insights into their working processes at that level of detail, and open sources systems rarely come with such extensive explicit artifacts and trace information.

A. Limitations

The evaluation process is exemplary of the processes at Frequentis, but doesn’t cover all of ED109. The model and engine however are not specific to ED109 and can be adopted to the specific process setting as shown with Dronology that followed a completely different process and TIM.

Adopting a different scenario then is mostly a matter of connecting different tools. Tools tend to come with a HTTP/REST interface, or client implementation (as did Jira and Jama with dedicated Java clients). Hence, it requires little effort in wrapping these clients for integration with the engine. New tools (and artifacts) are then accessible in the rules.

We also make the assumption that step completion can be detected from tools. The need for management, teamleads, and project leaders to obtain an accurate picture of progress, as well as having teams increasingly work distributed across multiple locations leads to a move away from informal signalling of completion toward explicit one, e.g., assigning a different member to an issue, setting a checkbox, setting the status of an issue, etc. Thus we believe that obtaining such indicators in almost all cases is reasonable.

VIII. RELATED WORK

Several researchers have proposed techniques for continuously assessing and maintaining software traceability [4]. EBT (Event Based Traceability) uses a publish-subscribe model to notify developers when trace links need to be updated [30].

| Table II: Dronology quality constraint evaluation results per process type |
|----------------|----------------|----------------|----------------|----------------|----------------|
| Task (n=135) | Sub-task (n=176) | Bug (n=95) | Design Def. (n=162) | Hazard (n=36) | SubWP (n=98) |
| ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| D-C1a | 44 | 91 | 32.6 | 94 | 100.0 | 100.0 |
| D-C1b | 8 | 94 | 82 | 53.4 | 96 | 100.0 | 100.0 |
| D-C2 | 135 | 0 | 100.0 | 164 | 8 | 95.5 | 96 | 100.0 |
| D-C3 | 132 | 3 | 97.8 | 164 | 12 | 93.2 | 98 | 100.0 |
| D-C4 | 94 | 1 | 98.9 | 136 | 26 | 84.0 | 26 | 100.0 |
| D-C5 | 94 | 4 | 95.9 | 136 | 26 | 84.0 | 26 | 100.0 |
| D-C6 | 94 | 4 | 95.9 | 136 | 26 | 84.0 | 26 | 100.0 |
| D-C7 | 94 | 4 | 95.9 | 136 | 26 | 84.0 | 26 | 100.0 |

| Table III: Quality constraint evaluation results per constraint type from completed process instances |
|----------------|----------------|----------------|----------------|----------------|----------------|
| Task % | Sub-task % | Bug % | Design Def. % | Hazard % | SubWP % |
| ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| AlwaysOk | 0 | 0.0 | 55 | 31.2 | 94 | 98.9 |
| EventualOk | 42 | 31.1 | 31 | 17.6 | 0 | 0.0 |
| CompleteNotOk | 93 | 68.9 | 90 | 51.1 | 1 | 1.1 |
| IncompleteNotYetOk | 53 | 82.8 | 22 | 62.9 | 10 | 71.4 |
| IncompleteProgressedNotOk | 11 | 17.2 | 5 | 14.3 | 0 | 0.0 |
| IncompleteOk | 0 | 0.0 | 8 | 22.9 | 4 | 28.6 |
| Total | 199 | 211 | 109 | 247 | 36 | 109 |
while Rempel et al., proposed an automated traceability assessment approach for continuously assessing the compliance of traceability to regulations in certified products [31], [32]. These approaches are orthogonal to our work as they are process-unaware, and hence provide little guidance for which step in a process a trace link must be available. Further, we assume engineers have chosen a suitable traceability strategy [33] and assessed that the resulting TIM (supported by flexible traceability management tools such as Capra [34]) also conforms to the relevant guidelines [2].

Process-centric software development environments (PCSDE) have received significant attention in the 90's. We discuss an exemplary selection below, for a detailed review see [35] or [36]. Step-centric modeling and active execution frameworks such as Process Weaver [10], SPADE [11], Serendipity [37], EvE [38], or PRIME [39] determine which steps may be done at any given moment, automatically executing them where possible. While such research supports detailed guidance, deviations from the prescribed process is not well supported. Approaches such as Shamus [40], PROSYT [41], or Merlin [42] specify for engineering artifacts which actions and conditions are available, and enforce their correct order — yet, without prescribing an overall step-based engineering process. Often the supported artifacts are limited to files and folders. Systems such as MARVEL [43], OIKOS [44], or EPOS [45] utilize ECA rules or pre- and post-conditions, thereby providing significant freedom of action to the engineer but offer limited guidance.

The approaches described so far have the implicit assumption that engineers primarily interact with the PCSDE for executing work. Our aim is to remain in the background, with engineers staying in their tools except for confirming QA constraint fulfillment. Provenance [46] has a similar goal, maintaining a process view from artifact change events. It’s, however, limited to events from the file system, relying on moving files to dedicated folders to signal process-meaningful events. It also remains unaware of trace links between artifacts.

More recent work focuses on specific aspects in the engineering life-cycle rather than general purpose processes. DevOpsML [47] aims at reducing the effort to describe continuous integration and deployment processes. Amalfitano et al. [48] aim to fully automate the execution of the testing process and to automatically generate appropriate traceability links. Similarly, Hebig et al. [49] investigate how various software design and code artifacts dependencies emerge from MDE activities. When involving human steps, approaches often assume pre-defined process models and rigorous tool integration. Kedji et al. provide a collaboration-centric development process model and corresponding DSL [50]. At a micro-level, Zhao et al. propose Little-JIL for describing fine-grained steps involved in refactoring [51] and to help developers track artifact dependencies during rework [52].

A few approaches on general purpose process modeling and execution (e.g., [53]-[56]) focus on step-centric languages such as SPEM and BPMN, which both imply active execution where engineers cannot deviate from the prescribed process. Business process compliance checking approaches determine whether complex sequences of events and/or their timing violate particular constraints. Ly et al. analyse frameworks for compliance monitoring [57] and highlight that the investigated frameworks have little or no inherent support for referencing data beyond the properties available in the respective events (hence no access to the actual artifact details and their traces/relations to other artifacts). They also show that hardly any approach supports proactive violation detection, the ability to continue monitoring after a violation, or root cause analysis in a manner useful for software engineering. Also very recent work, such as [58] or [59], lack this crucial support for defining constraints on artifact details. In general, processes in the software engineering domain are comparatively simpler but instead require a focus on keeping artifacts consistent with each other, hence data-centric constraints are required which our approach can check proactively and highlight that they are not yet fulfilled.

QA constraint checking exhibits some similarities to cross-artifact consistency checking. Examples include work on model-to-model [71], [60], [61] or model-to-code checking [6]. These approaches support the correct propagation of changes across artifacts once these artifacts are known to “belong” together. Our work, in contrast, supports the engineer in what state an artifact needs to be, and which trace links it needs to exhibit depending on the process progress.

IX. CONCLUSIONS AND OUTLOOK

We presented an approach for reducing the effort of ensuring that development activities adhere to quality constraints. The novel aspects are the decoupling of QA constraints from process control and dataflow. We thereby tolerate that engineers deviate from the process while informing them which constraints are yet unfulfilled and which steps are complete. Future work focuses on two main aspects. First we intend to study the effect of having our prototype in use by engineers at Frequentis. We aim to quantify the actual effort reduction and gather qualitative feedback for further improvements. Second, we will study QA engineers and process engineers during the creation, evolution, and maintenance of process models (including constraints) with ProCon to understand how their task can be supported even better.

X. DATA AVAILABILITY

The prototype and data used in this paper available at Figshare https://doi.org/10.6084/m9.figshare.12840053.

ACKNOWLEDGMENT

This work was funded by the Austrian Science Fund (FWF) under the grant numbers P31989 and P29415-NBL, and by the state of Upper Austria LIT-2019-8-SEE-118. The Dronology case-study was supported by the United States National Science Foundation under grants SHF:1741781 and CPS:1931962.
5.2 ProCon: An automated process-centric quality constraints checking framework

Authors:
Christoph Mayr-Dorn, Michael Vierhauser, Stefan Bichler, Felix Keplinger, Jane Cleland-Huang, Alexander Egyed, Thomas Mehofer

Venue:

DOI:
https://doi.org/10.1016/J.JSS.2023.111727 (open access)

Abstract:
When dealing with safety-critical systems, various regulations, standards, and guidelines stipulate stringent requirements for certification and traceability of artifacts, but typically lack details with regards to the corresponding software engineering process. Given the industrial practice of only using semi-formal notations for describing engineering processes – with the lack of proper tool mapping – engineers and developers need to invest a significant amount of time and effort to ensure that all steps mandated by quality assurance are followed. The sheer size and complexity of systems and regulations make manual, timely feedback from Quality Assurance (QA) engineers infeasible. In order to address these issues, in this paper, we propose a novel framework for tracking, and “passively” executing processes in the background, automatically checking QA constraints depending on process progress, and informing the developer of unfulfilled QA constraints. We evaluate our approach by applying it to three case studies: a safety-critical open-source community system, a safety-critical system in the air-traffic control domain, and a non-safety-critical, web-based system. Results from our analysis confirm that trace links are often corrected or completed after the work step has been considered finished, and the engineer has already moved on to another step. Thus, support for timely and automated constraint checking has significant potential to reduce rework as the engineer receives continuous feedback already during their work step.
ProCon: An automated process-centric quality constraints checking framework

Christoph Mayr-Dorn a,*, Michael Vierhauser a, Stefan Bichler a, Felix Keplinger a, Jane Cleland-Huang b, Alexander Egyed a, Thomas Mehofer c

a Johannes Kepler University, Linz, Austria
b University of Notre Dame, Notre Dame, USA
c Frequentis AG, Vienna, Austria

A R T I C L E   I N F O
Article history:
Received 21 March 2022
Received in revised form 18 April 2023
Accepted 24 April 2023
Available online 28 April 2023

Keywords:
Software engineering process
Traceability
Developer support
Quality assurance
Process deviation
Constraint checking

A B S T R A C T
When dealing with safety-critical systems, various regulations, standards, and guidelines stipulate stringent requirements for certification and traceability of artifacts, but typically lack details with regards to the corresponding software engineering process. Given the industrial practice of only using semi-formal notations for describing engineering processes – with the lack of proper tool mapping – engineers and developers need to invest a significant amount of time and effort to ensure that all steps mandated by quality assurance are followed. The sheer size and complexity of systems and regulations make manual, timely feedback from Quality Assurance (QA) engineers infeasible. In order to address these issues, in this paper, we propose a novel framework for tracking, and “passively” executing processes in the background, automatically checking QA constraints depending on process progress, and informing the developer of unfulfilled QA constraints. We evaluate our approach by applying it to three case studies: a safety-critical open-source community system, a safety-critical system in the air-traffic control domain, and a non-safety-critical, web-based system. Results from our analysis confirm that trace links are often corrected or completed after the work step has been considered finished, and the engineer has already moved on to another step. Thus, support for timely and automated constraint checking has significant potential to reduce rework as the engineer receives continuous feedback already during their work step.

© 2023 The Author(s). Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

1. Introduction

Software quality assurance (QA) focuses on ensuring and testing that the implemented engineering processes result in appropriate quality of the software. This not only includes code quality, but also pertains to the quality of the procedures, documentation, and available artifacts (Galin, 2004). To this end, various regulations, standards, and guidelines stipulate stringent traceability paths (Kramer et al., 2014; Rempe et al., 2014) without prescribing a corresponding, detailed software engineering process. Examples in safety-critical systems include the FDA principles in the medical domain (Code of Federal Regulations, 2021; McHugh et al., 2014), DO-178C/ED-12C for airborne systems (Brosigol and Comar, 2010), ED-109A (Jiménez et al., 2017) for air traffic management systems, and Automotive SPICE (Macher et al., 2017) in the automotive industry. To achieve compliance, QA engineers need to inspect fine-grained constraints related to properties of engineering artifacts, such as requirements, models, code, and test cases, as well as trace links at specific points in time (i.e., in different process steps, such as requirement elicitation, specification refinement, coding, or test case specification). The current practice in industry, however, is to employ semi-formal descriptions to define processes (Diebold and Scherr, 2017). As a result, there exists a crucial gap between the process model and the tool environment in which engineers (implicitly) enact the process. As a result, currently little to no automation support is available for engineers to understand whether they correctly follow a process, or to what extent they (temporarily) deviate from it.

In this work, we focus on the problems that Developers and Quality Assurance Engineers face when dealing with these processes, as adhering to, and evaluating QA constraints is complex and can quickly become overwhelming. Typically, developers work on multiple projects, sometimes simultaneously, with each project potentially adhering to different quality standards or guidelines.
We conducted an informal study with our industry partner Frequentis that applies the V-Model to develop (among other products) safety-critical air-traffic control software. Developers reported being stressed about potentially missing important steps mandated by quality assurance. A sub-problem, the challenge to correctly provide traces between engineering artifacts, is also commonly found in the automotive industry (Maro et al., 2018). QA Engineers, on the other hand, need to conduct countless, tedious, often mind-numbing checks that involve (manually) navigating across diverse artifacts and tools to ensure that the required constraints are fulfilled at the right process step. These checks are error-prone and rarely conducted in time to provide immediate feedback to developers. We observed that when quality checks are performed in batch for efficiency reasons towards the end of the development cycle, developers may only receive feedback as late as 6–12 weeks after completing their work (Mayr-Dorn et al., 2021). Remedying problems late in the process interrupts developers who may have already moved on to other steps or projects, causing disruptions and extra effort as they need to re-understand their past work context.

In this paper, we extend our prior work that was published at the International Conference on Software Engineering (ICSE’21) aimed at reducing the effort required in ensuring that development activities adhere to the intended process. Our approach provides automated support to developers and quality assurance engineers (Mayr-Dorn et al., 2021), and relies on passive process execution, i.e., by tracking process steps via monitoring engineering artifacts such as requirements, design documents, issues, change requests, and tests rather than engineers having to explicitly interact with a process engine. This is complemented by continuous evaluation of the specified quality constraints. The key novelty is treating (quality) constraints neither as an implicit part of the engineering process model, nor as completely disjoint from it. Instead, we propose treating quality constraint evaluations as first-class citizens: i.e., as explicit development artifacts that are used to monitor and determine process progress. This contrasts with existing work on traceability (Rempel et al., 2014; Cleland-Huang et al., 2014) where links required by regulations and standards are typically verified by an auditor at the end of a process stage or prior to shipping the final product (Watkins and Neal, 1994). Similarly, work on constraint checking (Egyed et al., 2018; Klare, 2018) primarily focuses on consistency among diverse artifacts without addressing consistency issues between these artifacts and the underlying process. Compared to past work on process-centric environments (PCEs) such as PRIME (Pohl et al., 1999a), our approach remains lightweight and requires minimal integration efforts — such human effort is one of the reasons past approaches were not readily adopted.

The key contributions of our work are as follows:

- A process model that decouples process control and data flow from QA constraints.
- A passive process engine that explicitly tolerates inconsistencies (Balzer, 1991) and allows engineers to temporarily deviate from the process while providing them with timely feedback on QA constraint evaluation results.
- A prototype that helps developers clearly understand when they have completed a step or what they still need to provide (e.g., specific content or trace links).
- An evaluation against an open-source system for unmanned aerial vehicles (UAVs), an industrial air-traffic control system (ATC), and a web-based recreational activity system that measures the extent to which the prescribed process was followed.

The prototype, process models, constraints, and historical development data used in this paper are available on Figshare.

The additional content, extending our original work, includes the following three main extensions: (1) an extended discussion on the background, as well as related work in the area, (2) an enhanced process model, and (3) significantly expanded evaluation including another industry evaluation case study and a preliminary usability study.

The remainder of this paper is structured as follows. Section 2 discusses the background of our work and motivates it by describing constraint checks for the development of safety-critical systems. Sections 4 and 5 provide an overview of our approach and introduce details on modeling the process and constraints, and Section 6 further focuses on constraint execution. We describe our evaluation method in 7, supported by three distinct case studies, and introduce our prototype implementation in Section 8 and evaluation setup in Section 9. We report details of the evaluation in Section 10, and conclude with a discussion of results (Section 11), threats to validity (Section 12), and related work (Section 3).

2. Background & motivation

Our work combines the areas of process management, traceability, and safety-related regulations. In this section, we provide a brief introduction to each of these areas and describe the context of our process engine using a motivating example from our industry collaborator.

Safety-critical software typically is subject to stringent regulations, where certain artifacts as well as trace links between the specification, individual requirements, test cases, and source code need to be thoroughly documented and provided during the certification process. For example, medical devices are subject to diverse international regulations, and software developed for the aerospace industry must comply with ISO/IEC12207 and DO-178C guidelines.

In the domain of Non-Airborne Systems, the DO-278/ED-109 standard (EUROCAE, 2012) defines requirements for traceability of artifacts, according to different design assurance levels. Creating, maintaining, and validating these links before certification is a costly and labor-intensive task that is typically performed manually, with little to no tool support. However, while DO-278/ED-109 describes the different types of assurance levels, ranging from “Catastrophic” to “No Effect”, and the types of trace links that need to be provided, it does not specify when these trace links need to be established or which role (i.e., who) should perform this task.

2.1. Motivating scenario

In this domain, of Non-Airborne Systems, our long-term industry collaborator Frequentis is a world-leading provider of voice communication solutions for air-traffic control and command-control centers. Their product portfolio in the air traffic domain ranges from aeronautical information management solutions, over remote digital towers, to traditional towers, with reliable voice communication playing a major role. Monitoring radio channels to obtain situational awareness, communicating with air traffic participants, and coordinating with other stakeholders in emergency situations introduce diverse use cases that all require real-time audio processing software and hardware, as well as the user interface, to work exactly as expected.

As part of their requirements engineering and development process, Frequentis refines high-level requirements and assigns

---

1 https://doi.org/10.6084/m9.figshare.12840053.
them to different work packages for engineers to work on. In order to meet the prescribed regulations, trace links need to be established between requirements, work packages, and the respective test cases. The left-hand side of Fig. 1 provides an overview of the partial traceability information model (TIM) used. The right-hand side provides excerpts from a simplified development process representation consisting of a sample set of process steps (S1 to S8) for one of their products. We use the described TIM, process, and constraints as a running example to describe our passive process engine framework.

In the example, high-level requirements (HLReq) and the corresponding high-level design specifications (HLSpec) are reviewed and subsequently refined into low-level requirements (LLReq). This, in turn, may require low-level design specifications (LLSpec) to be updated. Therefore, it is advisable for developers to wait for the outcome of the review before they start refining LLReq, even though HLReqs and teams are already assigned to work packages during the development iteration planning phase. Furthermore, before the implementation of LLReq can start, the corresponding trace links from HLReq to LLReq and trace links from LLReq to LLSpec need to be reviewed. In order to ensure adherence to the prescribed standards, constraints for the engineering process are introduced which rely on the properties of an artifact at a specific process progress. These not only check for the existence of an artifact or respective trace links, but further ensure that the correct links are set up between the right type of artifacts. For example, in the case of Frequentis’ development process, at the end of step S3 (cf. Fig. 1) the respective LLReqs need to be in state “Complete”, and contain a trace link to a HLReq. Furthermore, upon completion of step S7, when a LLReq’s verification method is “Demonstration”, at least one trace link to a respective Test Case must be established. This is further constrained by the fact that this test case must be a simulation, demonstration, acceptance test, or any other form of test, except the type “Software Test Case”.

Engineers who are updating the LLSpec (in S4), for example, thus need to be aware of when they can proceed with their step in order to avoid rework if additional refinements of an LLReq occur. Similarly, engineers in S7 need timely feedback when they can claim to have finished implementation and thus trigger the review in S8.

Knowing the state of the process helps assess the risk of deviation. Starting prematurely on HLReq refinement (S3) may be too risky if the requirements have not gone through review in S1 and S2 but perhaps necessary and expedient when S2 has only a few requirements left to review.

These constraints are not meant to replace human-in-the-loop QA measures, such as the trace reviews in S5 and S6, but to complement manual activities. Such (continuous) automated checks reduce the effort for reviews by ensuring the traces under review are syntactically correct and thus the review can be performed more efficiently.

2.2. Process definition vs. Process execution

As the scenario above exemplifies, in most software engineering environments, one cannot expect engineers to precisely follow the prescribed process definition. Various factors such as time pressure, unclear or missing information, or changing customer expectations, cause rework and make iterations necessary. Modeling all such possible “deviations” from an ideal process is often impractical and hence not done. The process, however, has an important guiding purpose, while additional QA constraints define properties of the engineering artifacts and their relations. The key aspect here is that strictly following the process is as detrimental to software quality as largely ignoring it. Hence, we make the case for having less detailed, but nevertheless, informative processes that are mapped to tools (in which they are executed), while tolerating deviations. This, however, should be no motivation to prescribe a waterfall model. Tracking process progress is crucial for engineers to assess whether an action leads to a deviation, or to assess who might be affected by a deviation. Tracking progress in the presence of deviations is crucial to understanding whether a deviation is critical, acceptable, and repairable. The next section discusses related work and its shortcomings with regard to supporting deviations.

---

2 Even W. Royce, the original author of the waterfall model (Royce, 1987), considers the execution of a strict waterfall process as risky and likely to fail.
3. Related work

3.1. Process-centric software development environments

Process-centric software development environments (PCSDE) have received significant attention in the '90s. We discuss an exemplary selection below, for a detailed review see Barthelmeiss (2003) or Grün (2002). Step-centric modeling and active execution frameworks such as Process Weaver (Fernström, 1993), SPADE (Bandinelli et al., 1996), Serendipity (Grundy and Hosking, 1998), EvE (Geppert et al., 1998), or PRIME (Pohl et al., 1999a) determine which steps may be done at any given moment, automatically executing them where possible. While such research supports detailed guidance, deviations from the prescribed process is not well supported. Approaches such as Shamus (LaMarca et al., 1999), PROSYT (Cugola and Ghezzi, 1999), or Merlin Junkermann et al. (1994) specify for engineering artifacts which actions and conditions are available, and enforce their correct order—yet, without prescribing an overall step-based engineering process. Often the supported artifacts are limited to files and folders. Systems such as MARVEL (Barghouti, 1992), OIKOS (Montangero and Ambriola, 1994), or EPOS (Conradi et al., 1994b) utilize Event Condition Action (ECA) rules or pre- and post-conditions, thereby providing significant freedom of action to the engineer but offering limited guidance.

The approaches described so far have the implicit assumption that engineers primarily interact with the PCSDE for executing work. Our aim is to remain in the background, with engineers staying in their tools except for confirming QA constraint fulfillment. Provence (Krishnamurthy and Barghouti, 1993) has a similar goal, maintaining a process view from artifact change events. It is, however, limited to events from the file system, relying on moving files to dedicated folders to signal process-meaningful events. It also remains unaware of trace links between artifacts.

More recent work focuses on specific aspects of the engineering life-cycle rather than general-purpose processes. DevOpsML (Colantoni et al., 2020) aims at reducing the effort to describe continuous integration and deployment processes. Amalfitano et al. (2017) aim to fully automate the execution of the testing process and to automatically generate appropriate traceability links. Similarly, Hebig et al. (2011) investigate how various software design and code artifacts dependencies emerge from MDE activities. When involving human steps, approaches often assume pre-defined process models and rigorous tool integration. Kedji et al. provide a collaboration-centric development process model and corresponding DSL (Kedji et al., 2012). At a micro-level, Zhao et al. propose Little-JIL for describing fine-grained steps involved in refactoring (Zhao and Osterweil, 2012) and to help developers track artifact dependencies during rework (Zhao et al., 2013).

A few approaches on general-purpose process modeling and execution (e.g., Dumais and Pfahl (2016), Eigner et al. (2010), Alajrami et al. (2016) and Winkler et al. (2019)) focus on step-centric languages such as SPEM and BPMN, which both imply active execution where engineers cannot deviate from the prescribed process.

3.2. Business process modeling

In the business process modeling domain, significant related work focuses on formally verifying processes (Morimoto, 2008) rather than attempting to fix them. Fixing is limited to achieving sound process models but is not applicable to instances as we aim for. The few, recent approaches that address inconsistencies and their repair exhibit limited expressiveness for specifying constraints: LTL for expressing task and event constraints (Maggi et al., 2011, 2014) or Mixed-Integer Programming for determining runtime compliance of task and resource allocation (Kumar et al., 2013). Business process compliance checking approaches determine whether complex sequences of events and/or their timing violate particular constraints. Ly et al. analyze frameworks for compliance monitoring (Ly et al., 2015) and highlight that the investigated frameworks have little or no inherent support for referencing data beyond the properties available in the respective events (hence no access to the actual artifact details and their traces/relations to other artifacts). They also show that hardly any approach supports proactive violation detection, the ability to continue monitoring after a violation, or root cause analysis in a manner useful for software engineering. Also, recent work such as Cabanillas et al. (2020) or Knuplesch et al. (2017), lacks this crucial support for defining constraints on artifact details.

Notably, the processes studied and used for evaluation in the business process management or information systems domain exhibit complex decision-making about which task to do next, or which task must not be done, and how much time between tasks may pass. Evaluation domains thus often include administrative processes, medical processes, or legal processes but virtually never software engineering processes. In the software engineering domain, processes are simpler but instead require a focus on keeping artifacts consistent with each other. Hence structural (i.e., data-centric) constraints are required which our approach checks proactively, subsequently highlighting that they are not yet fulfilled. Often the necessary guidance is not so much about which task to do next, but when to do it.

In comparison to dedicated software engineering process environments introduced above, general (business) process support, as provided by enterprise tools like SAP, is not applicable in software engineering environments as such support rigidly controls what steps may be worked on without any possibility of deviation.

3.3. Traceability

Several researchers have proposed techniques for continuously assessing and maintaining software traceability (Cleland-Huang et al., 2014). Event-Based Traceability (EBT) uses a publish–subscribe model to notify developers when trace links need to be updated (Cleland-Huang et al., 2003) while Rempel et al. proposed an automated traceability assessment approach for continuously assessing the compliance of traceability to regulations in certified products (Rempel and Mäder, 2016; Rempel, 2016). These approaches are orthogonal to our work as they are process-unaware, and hence provide little to no guidance for which step in a process a trace link must be available. Furthermore, we assume engineers have chosen a suitable traceability strategy (Rempel et al., 2013) and assessed that the resulting TIM (supported by flexible traceability management tools such as Capra (Maro and Steghofer, 2016)) also conforms to the relevant guidelines (Rempel et al., 2014).

3.4. Model consistency

QA constraint checking exhibits some similarities to cross-artifact consistency checking. Examples include work on model-to-model (Egyed, 2011; König and Diskin, 2016; Klare, 2018) or model-to-code checking (Egyed et al., 2018). These approaches support the correct propagation of changes across artifacts once these artifacts are known to “belong” together. Our work, in contrast, supports the engineer in what state an artifact needs to be, and which trace links it needs to exhibit depending on the process progress.
3.5. Research gap

The main research gap that we address in this paper is the lack of guidance in the presence of deviation from the intended process. Existing approaches either need to explicitly model the possibility to deviate or are too flexible (hence not providing sufficient guidance). Additionally, most approaches require engineers to interact with the process environment in order to track the process progress rather than a process environment passively observing the engineers’ activities in their tools and inferring process progress in the background from these activities.

4. Approach – The ProCon framework

In this section, we provide a comprehensive overview of ProCon, our framework for passive Process execution and quality Constraint support. Two of the key aspects that characterize our framework are the integrated handling of explicitly distinct processes and constraints, and the tracking of engineering progress realized through explicitly linking process descriptions to software engineering artifacts.

As part of ProCon, a passively executable Process Specification describes the sequence and alternatives of engineering activities (i.e., the “control flow”) and the corresponding software engineering artifacts serving as inputs and outputs of those activities (i.e., the “data flow”). Explicitly modeling a software engineering process for controlling the software engineering life-cycle is not new, with a plethora of research dating back to the 90s (Fernström, 1993; Bandinelli et al., 1996; Pohl et al., 1999b; Allou and Oquendo, 1998) (c.f. previous Section). ProCon is different insofar as (i) the process is tracked in the background, based on the software engineers’ activities performed in the tools they are using in their daily work, and as such, it does not require engineers to interact with a process engine, (ii) engineers are free to deviate from the process, (iii) engineers may receive guidance even in the presence of deviation, and (iv) ProCon supports control- and dataflow conditions as well as constraints across diverse artifact types and tools.

We refer to these abilities as “passive execution”. Instead of simply monitoring the process, our framework determines available future steps, detects premature steps, and makes this information immediately available to engineers. ProCon is capable of detecting and tracking deviations from a predefined process via a series of process and quality constraints. These constraints, and their respective evaluation results, are treated as first-class citizens in the software engineering process (model), and hence represent explicit software engineering artifacts in their own right. This in turn allows constraints to be explicitly evaluated as soon as actions are performed so that the evaluation results can provide valuable insights about the status of the process beyond whether a step has been completed or not.

4.1. Procon high-level architecture

ProCon consists of four major elements (depicted in Fig. 2) for defining, checking, and maintaining the process and development artifacts of an organization:

Existing Semi-informal process definitions, standards, guidelines, and regulations (A) serve as the initial input for ProCon. Typically, these already exist within an organization and describe (and motivate) the prescribed processes and quality assurance measures. Process definitions may include (parts of) the Software Process Engineering Metamodel (SPM) (SPM, 2008), PDFs representing flowcharts, or simple text documents outlining the steps and responsibilities of the different roles.

The process definition documents are complemented by – and used in conjunction with – a variety of diverse Tools within an organization to create, update, and maintain the artifacts that represent the input and output of the different process steps. Artifacts such as issues often serve as a (partial) informal representation of process instances. Tool Connectors (B) for the respective tools provide access to artifacts often in machine-readable formats such as JSON data. These connectors enable sophisticated tasks such as obtaining artifact updates via polling or subscriptions, and managing which artifacts are relevant for a process. It is, therefore, necessary to keep track of these artifacts, provide caching mechanisms for quick repeated access, and keep this cache up to date (cf. Section 8).

Passively Executable Process Specifications and Constraints (C) are manually derived (cf. Sections 5.1 and 5.4, respectively) based on the above two main inputs. Process Specifications formalize the (semi-)informal process definitions and guidelines and allow a fine-grained mapping to the respective artifacts, properties, and their changes observable via the tool connectors. The Passive Process Engine (D) manages instances of passively executable process specifications (or Process Instances for sake of brevity). The passive process engine obtains the state of an artifact and respective events from tool connectors (E), and feeds these changes into a rule engine in which the process progress conditions and QA constraints are evaluated. The rule engine eventually fires events that the process engine, in turn, utilizes to update the process progress and quality constraint evaluation results (cf. Sections 6 and 6.1, respectively).

Finally, a web-based Process Dashboard makes process progress and QA constraint evaluation results (F) continuously available to software engineers. It enables control of the passive process instance: triggering new instances, observing their progress, and eventually archiving them (cf. Section 8).

4.2. Procon usage

ProCon users are primarily QA engineers and developers, but may include other stakeholders, such as product managers and team leads. While the former can use the framework with a focus on quality assurance, which is the focus of this paper, the latter can leverage the framework to track and manage process progress.

Based on this, ProCon supports two distinct use cases. In the process and constraint modeling use case, various stakeholders map
the informal process definitions, standards, etc. (A) to passively executable process specifications (C). Stakeholders comprise dedicated process engineers, QA engineers, project managers, and others, depending upon how the organization assigns responsibilities for (i) defining how the development team needs to work to adhere to regulations and (ii) monitoring and improving these practices. Responsibility for providing a tool such as ProCon would fall under the responsibility of an IT department similar to maintaining other infrastructure such as source code repositories and issue tracking systems.

The definition phase involves analyzing how engineers currently produce evidence of process execution in the various tools (B). The tool connectors describe what artifact properties are available and thus what change events may serve as process progress triggers during process execution: for example, what (custom) properties are used for various Jira issues, or what trace links are available to navigate from a jira (2020) issue to a requirement managed in Jama (2020). The outcome of the modeling use case is a passively executable process specification and its quality constraints.

It is, however, important to note that our approach does not require modeling the complete process if tools do not facilitate access to certain information. One would typically start with those parts that are most important, or most error-prone, etc., and focus on these. Ambiguity may arise, for example, if rules expect one trace to navigate to an artifact but find multiple ones, then a random one will be selected. Mitigation includes correcting and/or extending QA constraints to identify these ambiguous situations. In line with Lee Osterweil’s key observation that “software processes are software, too” (Osterweil, 2011) we advocate the use of contemporary software engineering practices such as,

5. Process and constraint modeling

ProCon relies on a dedicated process meta-model that structures the constraint and artifact space. Furthermore, a number of process steps guide the (passive) execution of the process during the engineering life-cycle.

5.1. Process meta-model

One major challenge, when passively executing processes, is to determine the steps that are currently available for an engineer to work on, steps that represent work in progress (but perhaps should not be worked on yet), and finally, steps that have been successfully completed. In comparison to existing approaches, the major differences are not the basic building blocks (i.e., the process steps), but rather the way transitions between these steps are defined and subsequently triggered. Fig. 3 (left) provides a simplified UML class diagram of the main elements in the process specification and instance meta-model. The two main elements of the process model are Steps and the Decision Nodes attached to them.

A StepDefinition describes what an engineer “should” do (in contrast to “must” do – as prescribed by a more restrictive process). Examples include: “refine a requirement”, “implement a feature”, or “define a test case”. A defined step, in turn, has zero or more Input artifacts that represent data required for making a decision, creating a new artifact, or artifacts that need to be modified. Furthermore, zero or more Output artifacts are defined, describing the result of executing the step (e.g., having modified an input artifact or created a new artifact). Input and output artifacts can represent arbitrary kinds of information such as requirements, tests, issues, or trace links. While the StepDefinition only defines what type of Artifact is expected, identified via its Role, the InputArtifact and OutputArtifact in the process instance will contain the references to the actual artifacts of a concrete instantiated step. In addition to the textual description of an engineer’s activity, a step consists of a set of event-condition-action (ECA) rules that define which event(s) stemming from the engineering environment (e.g., an artifact update), given additional constraints (i.e., the condition), trigger the inclusion of an artifact to a step’s output artifact set (i.e., the action part of the rule). For example, in S5 “When an engineer posts a review URL as a Jira issue comment, then add that link as the step’s output artifact”.

A Decision Node describes how the completion of one or more Steps – and additional conditions – leads to the execution of subsequent steps. Therefore, the set of available decision nodes defines the control flow of the overall process. A decision node’s DataMapping declaration describes how the output of one step becomes the input of a subsequent step, thereby defining the process’s data flow. For example, “The LLReq output artifacts of S3 and LLSpec output artifacts of step S4 become the input artifacts to step S6”. As the WorkflowInstance is also a Step,
Beyond these standard building blocks, the process meta-model includes additional modeling elements to support our advanced use cases:

- **Constraining Step Input and Output**
  The main purpose of pre- and post-conditions of a step is to allow further refinement of input and output artifacts. The engine itself merely checks if there is at least one artifact available for each required input and required output. The step transition ECA rules may have the side effect of adding multiple artifacts per expected output instead of just one, and this is considered normal behavior. Step S3 HLReq Refinement into LLReq, for example, may specify that all refined LLReq are collected within one required output. In this case, a post-condition can ensure, for example, that there are at least as many refined LLReqs as there are provided Input HLReqs.

- **Reactivation of XOR steps**: The states COMPLETED, CANCELED, and NO_WORK_EXPECTED are not necessary final states. In the case of two or more steps from a set of XOR alternatives, the first step that becomes ACTIVE will cause the other step to transition to NO_WORK_EXPECTED. When the active step, however, becomes canceled, then the other steps now become relevant again and hence can transition from NO_WORK_EXPECTED back into AVAILABLE, ENABLED, etc. depending on the fulfillment of pre-conditions and firing of StepTransitionRules.

The same occurs if upstream changes (i.e., changes of artifacts in prior steps, thus process deviations) affect input artifacts in such a manner that a step becomes relevant again. Suppose we have two testing step types: unit test and manual test (that exclude each other) that become enabled depending on the validation type of their input requirement. If a delayed assessment causes the change of the requirement’s validation type, then the testing steps’ precondition evaluations will be inverted.

- **Deviations from expected state**: Aside from the expected task life-cycle FSM, we use the actual life-cycle FSM to track deviations between the defined and the actual process. For example, an engineer has performed a change but input artifact changes cause the pre-conditions to fail. Then the actual life-cycle FSM would transition into AVAILABLE (which is not possible in the expected FSM). This results in an explicit deviation whenever actual and expected life-cycle FSMs are not synchronous. Continuing with this example, eventually, this step might be not the selected one in an XOR, thus both FSMs transition into NO_WORK_EXPECTED, thereby resolving the inconsistency. Similar deviations are possible when a step is COMPLETED, CANCELED, or NO_WORK_EXPECTED but further work is observed once that state is reached. Here the actual FSM would transition into ACTIVE.

To distinguish when a transition to the active and completed state is deviating in contrast to repairing, we append ”_deviating“ and ”_repairing“ to the triggers in Figs. 4 and 5.

5.2. Process notation

The data model described above determines all information required by ProCon to instantiate a process. Our intention was not to mandate a particular visual notation, and we opted against using a preexisting language, such as BPMN (White, 2004), for two main reasons. First, we aimed to reduce the data model to only its core elements that are absolutely necessary. For example, BPMN defines far more elements than what is supported by our framework, hence making process design much harder for a process engineer, as they would need to recall which elements to use and which ones have no effect. Second, our constraints are defined as Drools rules and subsequently best written using a Drools editor, hence requiring the process engineer to switch between
several tools. However, as we define a canonical process format in JSON one could produce a transformation from a BPMN process model to our data model. This would be rather straightforward as noted above in Section 4: the key modeling elements themselves such as steps and AND, OR, and XOR decision nodes used for defining the process structure are not new. The novelty is in the way they are interpreted and allow for flexibility by ProCon.

5.3. Well-formed process specification

The process definition meta-model prescribes that a step must have exactly one predecessor decision node and exactly one successor decision node, but no further rules on what the emerging graph, containing these two element types, should look like. A well-formed process specification consists of a single kickoff decision node leading to steps that ultimately converge in a single-end decision node. Hence, only one decision node has no preceding steps, and only one decision node has no subsequent steps. Furthermore, whenever a decision node has \( k \) subsequent steps (with \( k > 1 \)), a decision node that “collects” these \( k \) branches (i.e., having \( k \) preceding steps) needs to exist at some point in the process. In the prototype implementation section, we show how our web-based process editor assumes responsibility for automatically ensuring such well-formed process specifications without burdening the process designer with this aspect.

5.4. Quality constraint integration

For each step, additional Quality Constraints can be defined, describing conditions for the newly created or updated output artifacts. Such a constraint can refer to all the information that is available for a step, such as its input and output artifacts, its local metadata, and its global process metadata.

Each constraint has an identifier that enables the triggering of constraints – not only upon artifact changes – but also manually, upon demand. It is important to provide control to the user, for example, by allowing an engineer to trigger a constraint check to reassure him/her that a step is indeed complete and nothing has been overlooked. ProCon does not prescribe a specific constraint engine or language in which constraints are defined, as this can vary depending on the domain and application scenario. The only requirement is that the adopted solution provides respective evaluation results that can be further processed. Further examples of constraints and the constraint language are discussed as part of the case studies described in Section 9).

Every time a quality constraint is evaluated, a corresponding new instance of Quality Constraint Evaluation Result is created. This evaluation result not only reports the result (i.e., fulfilled or unfulfilled) of the evaluation, but also lists the artifacts affected by the constraint. For example, a constraint for step S6 checks whether each low-level requirement (LLReq) output artifact has a trace link to a high-level requirement (HLReq) which must be in state “released”. Therefore, the respective constraint evaluation result will contain the list of LLReqs that fulfill this constraint, and a list of LLReqs that violate this constraint. For each constraint, ProCon further collects timestamps indicating when the constraint evaluation result was last evaluated, and when the evaluation result last changed from being violated to fulfilled and vice versa. This, for example, allows a user to determine how recent the results are when displayed in the Process Dashboard. The evaluation results of all quality constraints associated with the same step are collected and bundled into a Quality Check Document which is added to the output artifacts of that step.

The process model itself remains largely independent from constraints and their evaluation results. While a process step’s completion constraint must check whether a Quality Check Document output artifact exists which contains only positive Quality Constraint Evaluation Results, it does not need to understand the particular constraints that resulted in the evaluation success.

This rather loose coupling allows the same process to be executed with different levels of quality assurance by simply replacing quality constraints. Inversely, a quality constraint may be used in different process contexts when the respective process step provides the artifacts on which the constraint is evaluated.

5.5. Supporting semi-structured artifact properties

Development tools quite frequently support artifacts that cannot be sufficiently customized to match the underlying development process. In such cases, users typically revert to providing information in text fields or rich edit fields where data is entered using HTML tags or markdown. To this end, the process engine utilizes the full potential of Java that comes with defining the rules and constraints in the Drools rule engine. ProCon supports the extension of individual artifacts with arbitrary key–value pairs that are populated when an artifact changes based on custom drools rules. These rules transform semi-structured information from the artifact to structured key–value pairs that

---

3 We are currently working on unifying the process structure and constraints to remove the need for tool switching.
can then be used in pre-conditions, post-conditions, StepTransitionRules, etc., hence separating the concern of how to extract information from artifacts from the concern of processing that information.

6. Passive process execution

Every process specification model comes with a set of required input artifacts, typically representing a change request issue or work package issue that already serves as some form of informal process representation. There are three ways in which a process instance can be created. First, manually when a ProCon user provides the identifier(s) of the input artifact(s) via the framework’s user interface. Second, when a subprocess step definition is instantiated, the step instance then creates the corresponding process passing its input artifacts along. Third, when a rule in an existing process instance spawns a new separate process. The difference between the second and third option is that in the former case, the framework ensures that output from the subprocess is mapped back into the subprocess step and any relevant state changes of the subprocess step such as canceling are propagated to the subprocess. In the latter case, the spawned process remains completely independent from the spawning process. The last option is particularly useful for creating multiple fine-grained processes, for example, one for each story within an epic.

The process engine instantiates the process and triggers the initial decision node instance, which in turn instantiates the first step(s) and executes the data mappings from process input(s) to step input(s). Further instantiation happens incrementally, only when a step reaches the state ENABLED, and the process engine instantiates the step’s subsequent decision node(s). Similarly, only when a decision node’s context condition is fulfilled, the engine instantiates the subsequent steps. Step instantiation happens incrementally, only when the decision node’s context condition is fulfilled, the engine instantiates the subsequent steps. With step instantiation, the engine also instantiates the quality check constraints. Thus, as long as a step does not exist, none of its constraints will be checked.

However, as passive process execution does not mandate and enforce a fixed, predefined step, incremental step and decision node instantiation are not sufficient in our case. For example, when an engineer decides to work on a step that is not yet available, as soon as the artifact changes trigger StepTransitionRules, the process engine must instantiate the corresponding “premature” step and retrieve an existing preceding decision node that the stepption rules can be linked to. If no preceding decision can be found, the step remains dangling in the process until the process progress catches up: i.e., upon instantiating a decision node, the engine checks if a dangling step exists that should be linked from. From the engine’s perspective, there is no difference between missing a step and starting the next, or starting too early on the next step. It will continue either way. In such a case, however, the engine will not be able to fully execute a DataMapping that requires the output of the skipped/incomplete step. The consequence is that many tools do not provide an active notification mechanism when changes to artifacts occur. For example, does not offer automatic event notifications, but requires polling with subsequent explicit fetching of changed artifacts. In the case where polling intervals cannot exceed a certain duration, to avoid introducing performance issues, engineers will not immediately notice effects of their work on QA constraints. This effect becomes even more severe when a single change is not indicative of step completion or QA constraint fulfillment. This is the case when quality constraints span across multiple artifacts and potentially even multiple tools: a single change to an artifact then is insufficient, multiple changes need to occur. For example in S6: not just one low level requirement (LLReq) needs to be set to “released” but all linked ones. In such a situation, on-demand fetching of updates ensures that the engine that all QA constraint evaluations occur on the most recent artifact versions, and any violation will not be due to stale data.

6.1. Constraint checking

ProCon provides two possible ways of how constraint checks can be performed. By default, QA constraints and StepTransitionRules are evaluated upon every single artifact change. As ProCon tracks which process instance accesses which artifacts and forwards the respective change events, only constraints relevant for the process instance needs to be checked.

Additionally, we allow for manual user input, enabling an engineer to request explicit checks of artifact updates and subsequent evaluation. The reason for this is based on the fact that many tools do not provide an active notification mechanism when changes to artifacts occur. For example, does not offer automatic event notifications, but requires polling with subsequent explicit fetching of changed artifacts. In the case where polling intervals cannot exceed a certain duration, to avoid introducing performance issues, engineers will not immediately notice effects of their work on QA constraints. This effect becomes even more severe when a single change is not indicative of step completion or QA constraint fulfillment. This is the case when quality constraints span across multiple artifacts and potentially even multiple tools: a single change to an artifact then is insufficient, multiple changes need to occur. For example in S6: not just one low level requirement (LLReq) needs to be set to “released” but all linked ones. In such a situation, on-demand fetching of updates ensures the engine that all QA constraint evaluations occur on the most recent artifact versions, and any violation will not be due to stale data.

6.2. Propagating artifact input/output changes

In the same manner as StepTransitionRules monitor artifacts for changes to add output artifacts (e.g., a requirement gets linked to a test case), they remove output artifacts when their conditions are no longer met (e.g., a link to a test case is removed again). The process engine ensures that such artifact usage changes are properly propagated to subsequent tasks. For example, in a case where S4 (Updating HLSpec) is completed and S5 (HLReq to LLReq Trace Review) has started. Now deviating from the process by having in the scope of S4 adding another trace between a HLReq and LLReq. Execution of S4’s StepTransitionRules would result in another trace (also an artifact) to be added to S4’s output which needs propagating as input to S5. Likewise, removing such a trace

---

4 How to best aggregate and display meaningful percentage data is outside of the focus of this paper, and part of future work.

5 We acknowledge that highly dynamic, highly iterative development environments perhaps might not benefit as much from our framework.
6.3. Rule execution order

The majority of project/team-specific aspects such as task preconditions, post-conditions, StateTransitionRules including adding output, and quality constraints are defined as rules which in turn signal changes to the process. Together with the process engine internal change triggers such as task instantiation and DataMappings etc., these change evaluation sources need to be activated in a predetermined order to avoid “race” conditions: e.g., executing post-condition checks that signal task completion before adding new output that would cause these post-conditions to fail and hence not to signal task completion. Recall that the ultimate source of any activity in the process engine is an artifact change. Upon an artifact change, the change evaluation sources then are evaluated in the following order⁶:

- Adding input to tasks
- Pre-processing of artifact change events (e.g., when semi-structured artifact properties need to be parsed)
- StateTransitionRule which evaluate the pre-conditions
- StateTransitionRules which add output to tasks
- StateTransitionRules which activate or cancel the step
- Quality constraint evaluation
- StateTransitionRules which complete the step (i.e., the post-conditions)

7. Evaluation method

Evaluating our ProCon framework, we investigate five research questions regarding the feasibility, flexibility, performance, and usability of our approach. Specifically, we assess the general feasibility of our approach by evaluating it against real historical process data from three distinct case studies. For each case study, we created process specifications and constraints, and implemented connectors for issue tracking and requirements management tools.

7.1. Research questions

While this research project was originally motivated by a pain point of our industrial collaborators (Frequentis), our first two research questions investigate the nature of process deviations and constraint violations (i.e., from a process view and a QA constraints view, respectively) to empirically evaluate the need for a tool such as ProCon.

RQ1: How frequently do process deviations occur, and to what extent are these temporary? Rationale: As part of this research question we investigate process deviation problems to uncover whether engineers continue their work even in the presence of QA violations, and ultimately the extent to which they require support to eventually fix these process deviations (i.e., investigating from a process perspective). To this end, we replay historical engineering process events and establish if and when engineers continued to a subsequent step in the presence of a constraint violation.

⁶ Note that typically only one or two such change evaluation sources become indeed active and trigger a consequent change such as a step state change.
Cleland-Huang and Vierhauser, 2020). For the purpose of this case study, we obtained permission to use data from multiple sprints maintained in Jira from 2017 to 2019. This includes the following artifacts: Bugs, Hazards, Requirements, Design Definitions, Tasks, and Sub-Tasks.

**CS-2: Frequentis**

For the Frequentis case study, we selected a safety-critical product for voice communication in air traffic control centers. The product consists of several different subsystems for interfacing with radio transceivers, managing near-real-time voice streams, and providing operator user interfaces. Frequentis follows a V-model (Verein zur Weiterentwicklung des V-Modell XT, 2021) like engineering process. Specifically, for this case study, we focused on sub work packages (SubWP). This process for each team (each responsible for one subsystem) starts with high-level requirements resulting in the actual implementation and the successful execution of test cases. This covers steps S3 to S8 of the motivating scenario. Trace links between SubWP’s and low-level requirements are therefore the main focus of QA constraints defined as the completion conditions of steps S3 (represented by ATC-C1 to ATC-C4), S4 (ATC-C5), S7 (ATC-C6 and ATC-C7), and S8 (ATC-C8) (cf. Table 2). Frequentis uses Jama to manage all artifacts and trace links depicted in Fig. 1 and uses Jira to manage the engineering process.

**CS-3: ACME-RA**

ACME-RA7 is in the business of hosting a recreational activities web platform. ACME-RA tracks development progress with Jira, heavily customizing available issue states and state transitions. Different issue types come with different states and transitions. Based on feedback from a developer at ACME-RA we selected issues of type “Task” as a representation for a non-trivial development sub-process. ACME-RA follows an agile development methodology, pulling issues in from a backlog for each sprint. Issues then undergo a set of possible transitions some of which require a particular engineering role to be involved. Once a new task is created, its initial state is Open. The state changes to In Development as soon as the work on the issue starts. After finishing the task the developer changes the state to Ready for Review when needed. A quality assurance engineer then picks up the task and assesses the issue before changing the state to Reviewed. Alternatively, no review is conducted and the task is regarded as finished, hence it changes to Resolved (one of the allowed end states). If required, testing can be performed after resolving the task (state In Testing). Both development and testing can be suspended (states Suspended Development and Suspended Test), for example, in case additional resources are required, and then resumed when these resources become available. Additional states track when a task needs to be in development again, is suspended again, or reviewed again.

**CS-4: Siemens L+A**

The case study from Siemens Logistics and Automation, described by Cleland-Huang et al. (2007), involves various artifacts stakeholders use to capture the requirements and system components for automatically obtaining a shop floor layout. Business Goals (BG), Stakeholder Requests (SR), Minimal Marketable Features (MMF), and Business Use Cases (BUCs) are primarily used by business end users, while developers primarily interact via System Use Cases (SUCs), Concrete System Capabilities (CSCs), and Concrete System Components (CSComp). While a significant set of traces exist, the process of implementing a stakeholder request can be modeled rather straightforward as a sequence of

7 The company’s identity and project names have to remain confidential due to the sensitive nature of the analyzed data.

"WriteOrReviseMMF", "RefineToSUC", and "CreateOrRefineCSC" steps.

**Regulatory Certification**

The case studies are subject to different regulatory certifications. The Dronology project (CS-1) aims to follow best practices and guidelines as, for example, specified in DO-178C (Bros gol and Comar, 2010) but does not require official certification, and hence is not certified. The ATC system by Frequentis (CS-2) is developed to be compliant with ED-109A/DO-278A (Jiménez et al., 2017) and externally certified. In our evaluation, we focus mostly on the respective standards’ traceability requirements, whereas the process description stems from engineers at the two case study organizations, describing how they manage software development and fulfillment of traceability constraints. ACME-RA (CS-3) and Siemens L+A (CS-4) are not subject to certification but were specifically chosen to demonstrate the applicability of our framework in non-regulated domains.

**Differences in Case Study Results Reporting**

Note that these case studies serve different evaluation purposes. CS-1 Dronology, CS-2 Frequentis, and CS-3 ACME-RA provide real historic process data to answer RQ1 and RQ2. While CS-2 has a simple underlying process but complicated constraints, CS-3 additionally was chosen to evaluate whether ProCon can also support a more complicated process where engineers repeat steps and may choose to skip steps (RQ3) but CS-3 yields more simple constraints than CS-2.

Performance evaluation in the scope of RQ4 only considers CS-2 and CS-3 as the former comes with the most complicated constraints among the four case studies and the latter comes with the most complicated process progress constraints among the four case studies. Both case studies come with a significant amount of related artifacts and change events. Hence they produce realistic loads on ProCon.

Finally, CS-4 Siemens L+A was selected for evaluating ease of use of specifying process and QA constraints as it comes with a reasonably complicated process structure, together with reasonably complicated QA constraints — thereby representing a middle ground between CS-2 and CS-3. As we have no access to historical process data for CS-4, we cannot report results for RQ1 to RQ4 for this case study.

**8. Prototype implementation**

To support the evaluation, and to obtain feedback from our industry partner Frequentis about our approach and the provided tool support, we created a prototype implementation of ProCon.

**Tool Connectors:** To cover a diverse set of artifacts from our industry partner and from the Dronology case study, we implemented connectors for Jira, a web-based tool for planning, issue tracking, and reporting, and Jama, a tool for requirements management, traceability, and test management. The Jira Connector uses the Atlassian Java REST API to retrieve artifacts and their attributes, and is used to periodically poll for changes in these artifacts. Similarly, the Jama connector uses the Jama REST API. To reduce the load on network and tools, the tool connectors cache Jira and Jama artifacts in a MySQL database in their native JSON format as obtained from the REST interface.

**Process Engine:** The Process Engine is implemented in Java, containing an implementation of the process specification meta-model and a rule engine for checking constraints. We opted for the Drools (Drools, 2020) rule engine, a Business Rules Management System that can be easily integrated into a Java application and allows easy access to Java objects (representations of Jira and Jama artifacts) within rules written in a Java dialect. The process engine is wrapped in a web application (see Fig. 6) using the
we provide a preliminary visual process editor (see, for example, a screenshot of the ACME-RA process in Fig. 8) utilizing Blockly.\(^9\) The editor ensures the well-formedness of the process steps, and generates the boilerplate code from the visual representation. In the rules excerpt of Fig. 7 only lines 181 to 183 out of lines 271 to 187 had to be manually added, with the rest being generated automatically by the process editor. Overall, the process progress rules for the ACME-RA case study involved around 600 lines of code (not counting any Java imports), out of which only around 70 had to be manually written. Similar for the quality constraints: for the four ACME-RA constraints, only 14 lines out of 150 (again without Java imports) had to be manually defined.

The preliminary process editor enables a process engineer to define the steps, the types of artifacts used in input and output, their parallelism/alternatives, and which artifacts are utilized in StepTransitionRules and QA Constraints. Formulating the actual conditions of the rules and constraints is not yet supported as we are still investigating which type of visualization is best suited to specify complex conditions over artifact properties. Nevertheless, this structure is sufficient to generate all boilerplate code.

**Process Dashboard:** Fig. 9 shows the user interface for inspecting quality constraint evaluation results. The results contain links to the original artifacts, enabling engineers to quickly switch to their commonly used tools (here Jama and Jira) to investigate and fix any unfulfilled constraints. The process dashboard is automatically updated whenever a step, decision node, or quality constraint evaluation changes without the user having to poll for updates in the browser. There the user also has the option to inspect available or missing artifacts and even manually add artifacts to step input and output in case the currently active rules fail to identify these from the process context.

**9. Evaluation preparation**

Before ProCon can be used with a specific process and system, a number of preparatory steps are necessary, to identify, select, and integrate artifacts and tools, and create respective constraints. In the following, we briefly describe the relevant steps for our four case studies.

**9.1. Evaluation setup**

**CS-1: Process and Constraint Creation:** For the purpose of this case study, we treat each of the collected Droconly issues as "small sub processes". The state of each issue represents a process step, and quality constraints for each step describe the conditions that need to be fulfilled to transition from one step to the next and to complete the process (i.e., close the issue). Given the lean nature of typical agile open-source development processes, the states observed are limited to the default process steps in Jira, "Open", "InProgress", and "Closed". Based on the information

---
\(^8\) https://axoniq.io.  
\(^9\) https://developers.google.com/blockly.
available, we identified the following eight quality constraints and allocated them to the steps where they are most useful (note that some constraints are reusable for multiple issue types). Defining the processes and constraints used in this evaluation took approximately three hours and did not include familiarizing with the project’s artifact types and traceability strategy. We then confirmed the validity of the constraints and the process together with the lead developers of the project. An overview of the constraints can be found in Table 2. At the end of step “Open”, we require constraints D-C1 to D-C5 to be fulfilled, and at the end of step “In Progress” we require constraints D-C6 and D-C7 to be fulfilled.

CS-2: Process and Constraint Creation: For the second case study, we defined the eight constraints from the SubWP process together with a QA engineer from Frequentis. This was done in approximately two hours, and another two hours were spent extracting process information from documents and specifying the actual process. This duration did not consider familiarizing with the artifact types and their traces as this was done earlier in the collaboration with Frequentis and can be assumed to be common knowledge of a QA engineer. Frequentis’ informal process definition precisely defines how engineers need to set properties of Jira and Jama artifacts for completing the various steps. Changes to these properties serve as step-completion signals in our process engine.

CS-3: Process and Constraint Creation: For the third case study, the web platform, we selected “Task” issues and transformed the state transitions into a process consisting of seven steps (see Fig. 10). Any issue state that indicates a repetition of some activity was treated as a reactivation of the corresponding process step. For example, reopening an issue (i.e., Reopened) is interpreted as a reactivation of the “Prepare” step. Together with a developer from ACME-RA we discussed the conditions under which these transitions should occur and encoded three of them as quality assurance constraints (listed in Table 2 bottom). 10 Constraint RA-1 needs to be fulfilled at the end of step “Preparing”, constraint RA-2 is required only in case a review is conducted (i.e., step “Reviewing”), constraint RA-3 is required only in case testing is carried out, and constraint RA-4 is checked for finalizing the “Completing” step. The four constraints were straightforward to encode within half an hour, while the step activation, completion, etc. rules required more time (around 4 hours), due to the extensive use of different issue states.

For the purpose of this paper, in the ACME-RA case study, we focus on demonstrating (aside from QA constraint checking) the ability of the passive process engine to handle revoked, reactivate, canceled, and prematurely started process steps.

CS-4: Process and Constraint Preparation: In a first step, we mapped the TIM to Jira by creating a dedicated Jira issue type for each artifact type, and mapped the process to a Jira story with the individual steps modeled as that story’s subtasks. We then created example artifacts and traces (including an example process instance). Such a grounding of the TIM and process in actual tools is required in any case, independent of the application of ProCon.

Given the simplicity of the process compared to the more complex traceability information model, the preliminary usability evaluation focuses on the effort required for specifying process step transition rules and the QA constraints. To this end, we prepared the process specification with the ProCon process editor and generated the Drools templates for the process steps transitions and quality constraints.

Performance measurement We focus on measuring the time ProCon requires to check all constraints after an update event (i.e., a change to an artifact) occurred. We leave aside any time required to load the artifacts from their originating tools as this is heavily influenced by the tool’s API 11 current load and availability of artifacts already in ProCon’s cache. Hence, we measure the time from replaying the first of the change events to the end of processing the last change event for CS-2 Frequentis and CS-3 ACME-RA. We measured this interval for 10 replay runs on a standard Core i7 laptop, with 8 GB of RAM to obtain the average duration. Dividing the duration by the number of relevant events (i.e., those that potentially affect the constraint evaluation result) provides insights into the expected average processing time to evaluate one change event. We measure in the presence of multiple process instances as realistically a change event potentially affects multiple constraints (in the scope of multiple process instances) that all need separate re-evaluation.

User Experiment Preparation In the scope of the small, controlled experiment, we asked six software developers that had not used ProCon before to implement the StepTransitionsRules for activation and completion for each process step as well as the input to output DataMappings (experiment task 1) and to implement

---

10 Due to the limited amount of issue details, we were only able to encode a small subset of the actual constraints.

11 Tools differ in the number of API calls required to obtain artifact details and artifact updates.
the six QA constraints textually given in the process specification (experiment task 2) by filling out the generated Drool templates. Overall, the participants had to write six StepTransitionRules, seven DataMapping rules, and six QAConstraints.

We recruited six participants, each of whom had some knowledge about software processes and basic knowledge about Drools, but had not used ProCon before to write quality constraints or process progress constraints. The participants had between 1 and 6 years of Java development experience in the scope of research project employment at our research institution.

The six participants had access to the ProCon development environment in Eclipse comprising the Drools Editor, ProCon Java API, with source code to access and inspect Jira issues, as well as code for executing their process and constraints rules. We additionally provided one example process step transition rule and one QA constraint from CS-3 as a reference.

During the 75 min experiment, the participants received a 15-min introduction to the framework and development environment. They then had 30 mins and 20 mins available for the two experiment tasks, respectively, and 10 mins to provide informal feedback at the end.

During the experiment, we documented how much of the task each participant was able to complete, and how many mistakes they made that remained in the final output. Additionally, participants received minor feedback on the Jira Artifact’s API.

9.2. Data gathering

For the Dronology project, we received access to the Jira server REST API to obtain artifacts and their change history. The data set consists of 802 process instances (i.e., Jira issues): 199 Tasks, 211 Sub-tasks, 109 Bugs, 247 Design Definitions, and 36 Hazards.

From Frequentis we obtained Jira issues related to the aforementioned SubWPs. Each SubWP managed in Jira has a corresponding Jama artifact with respective trace links to LLReqs and subsequent artifacts. We used the Jama REST API to navigate across these trace links to collect all Jama artifacts (including their history) that are relevant for constraint evaluation. This resulted in a set of 109 SubWPs and ~14,000 linked Jama items (out of which 1121 are LLReqs).

From ACME-RA we received a JSON dump of Jira issues of four multi-year projects: P1, a low-priority Android app development project; P2 and P3, two business-critical Android App development projects; and P4, a project integrating two types of recreational activities that involved experts beyond front-end, business logic, design, and database engineering (e.g., marketing and legal departments). In total the dataset contained 1017, 2676, 1052, and 939 issues, respectively. The five most common issue types were “Task”, “Bug”, “Improvement”, “Localization”, and “Project Management” and make up between 80% and 90% of all issues. All issues had their change history reduced to changes to the properties assignee, state, fix version, and due date. Additionally, the anonymized user identifier were enhanced with role identifiers to distinguish between front-end developers, back-end developers, database developers, team leads, quality assurance engineers, graphics designers, marketing engineers, and bots. Project P1 was the first to start, its end interleaving with P2 and P3 which had a similar duration, their end again interleaving with the start of P4 which, at the time of data gathering, was not completely finished yet.

For this study, we retained only issues of type “Task”, that were successfully resolved (i.e., in state “Fixed”), with a non-empty set of child issues. This resulted in 46, 21, 119, and 81 process instances, respectively.
Table 2
Constraints derived for the Dronology use case ("D"), the Frequentis ("ATC"), the ACME-RA ("RA"), and the Siemens L + A ("LA").

<table>
<thead>
<tr>
<th>Constr.</th>
<th>Description</th>
<th>Issue type</th>
</tr>
</thead>
<tbody>
<tr>
<td>D-C1a</td>
<td>The issue traces to one or more Design Definitions</td>
<td>Tasks</td>
</tr>
<tr>
<td>D-C1b</td>
<td>The issue traces to one or more Design Definitions directly, or via its parent</td>
<td>Task, Sub-Task</td>
</tr>
<tr>
<td>D-C2</td>
<td>The issue does NOT trace to a requirement</td>
<td>Task, Sub-Task</td>
</tr>
<tr>
<td>D-C3</td>
<td>The issue has an assignee</td>
<td>Bug, Task, Sub-Task</td>
</tr>
<tr>
<td>D-C4</td>
<td>The issue traces to a requirement</td>
<td>Design Def.</td>
</tr>
<tr>
<td>D-C5</td>
<td>The issue is mitigated by a requirement (i.e., trace type: isMitigated) or refined by a Hazard (i.e., trace type: isRefined)</td>
<td>Hazards</td>
</tr>
<tr>
<td>D-C6</td>
<td>The issue has all related bugs (if any) closed</td>
<td>Task, Sub-Task</td>
</tr>
<tr>
<td>D-C7</td>
<td>The issue has all sub-tasks (if any) closed</td>
<td>Bug, Task</td>
</tr>
<tr>
<td>ATC-C1</td>
<td>All traced LLReq have status &quot;released&quot;</td>
<td>SubWP</td>
</tr>
<tr>
<td>ATC-C2</td>
<td>All traced LLReq have a release assigned</td>
<td>SubWP</td>
</tr>
<tr>
<td>ATC-C3</td>
<td>All traced LLReq have a trace link to at least one HLReq</td>
<td>SubWP</td>
</tr>
<tr>
<td>ATC-C4</td>
<td>No traced LLReq has a trace to another SubWP with a status other than &quot;closed&quot;</td>
<td>SubWP</td>
</tr>
<tr>
<td>ATC-C5</td>
<td>All traced LLReq have a trace link to exactly one Functional Unit</td>
<td>SubWP</td>
</tr>
<tr>
<td>ATC-C6</td>
<td>All traced LLReq have a link to at least one test case matching the requirement's verification method</td>
<td>SubWP</td>
</tr>
<tr>
<td>ATC-C7</td>
<td>The SubWP's Jira issue has at least one &quot;Fix version&quot;.</td>
<td>SubWP</td>
</tr>
<tr>
<td>ATC-C8</td>
<td>The SubWP's Jira issue is set to &quot;resolved&quot;</td>
<td>SubWP</td>
</tr>
<tr>
<td>RA-C1</td>
<td>An engineer of role 'developer' needs to be the issue assignee</td>
<td>Task</td>
</tr>
<tr>
<td>RA-C2</td>
<td>An engineer of role 'QA' conducts the review</td>
<td>Task</td>
</tr>
<tr>
<td>RA-C3</td>
<td>An engineer of role 'QA' conducts the test</td>
<td>Task</td>
</tr>
<tr>
<td>RA-C4</td>
<td>All sub-tasks are in state 'Closed'</td>
<td>Task</td>
</tr>
<tr>
<td>LA-C1</td>
<td>Each Process (story) must trace to at least one SR via a 'Realized' link</td>
<td>Story</td>
</tr>
<tr>
<td>LA-C2</td>
<td>Each linked MMF must trace to at least one SR via a 'Realizes' link.</td>
<td>Subtask</td>
</tr>
<tr>
<td>LA-C3</td>
<td>Each linked MMF must trace to at least one BUC via a 'Realizes' link.</td>
<td>Subtask</td>
</tr>
<tr>
<td>LA-C4</td>
<td>Each SR linked via an MMF must trace to at least one BUC via a 'Relates' link.</td>
<td>Subtask</td>
</tr>
<tr>
<td>LA-C5</td>
<td>Each SUC must trace to at least one (parent) SUC or a BUC via a 'Realizes' link.</td>
<td>Subtask</td>
</tr>
<tr>
<td>LA-C6</td>
<td>Each CSC must trace to at least one SUC via a 'Realizes' link.</td>
<td>Subtask</td>
</tr>
</tbody>
</table>

Table 3
Quality constraint evaluation results per process type (Dronology case study).

<table>
<thead>
<tr>
<th>Process</th>
<th>Task %</th>
<th>Sub-task %</th>
<th>Bug %</th>
<th>Design Def. %</th>
<th>Hazard %</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlwaysOk</td>
<td>0</td>
<td>0.0</td>
<td>55</td>
<td>31.2</td>
<td>94</td>
</tr>
<tr>
<td>EventualOk</td>
<td>42</td>
<td>31.1</td>
<td>31</td>
<td>17.6</td>
<td>0</td>
</tr>
<tr>
<td>CompleteNotOk</td>
<td>93</td>
<td>68.9</td>
<td>90</td>
<td>51.1</td>
<td>1</td>
</tr>
<tr>
<td>IncompleteNotYetOk</td>
<td>53</td>
<td>82.8</td>
<td>22</td>
<td>62.9</td>
<td>10</td>
</tr>
<tr>
<td>IncompleteProgressedNotOk</td>
<td>11</td>
<td>17.2</td>
<td>5</td>
<td>14.3</td>
<td>0</td>
</tr>
<tr>
<td>IncompleteOk</td>
<td>0</td>
<td>0.0</td>
<td>8</td>
<td>22.9</td>
<td>4</td>
</tr>
<tr>
<td>Total</td>
<td>199</td>
<td>211</td>
<td>109</td>
<td>247</td>
<td>36</td>
</tr>
</tbody>
</table>

We used our trace link replay tool (Mayr-Dorn et al., 2020) to reset all datasets, particularly the Jira issues and Jama items and their trace links to the earliest change event and then replayed every single change in the correct temporal order. The changes occurred between April 2017 and December 2019 for Dronology, between May 2018 and June 2020 for Frequentis, and between December 2013 and January 2018 for ACME-RA, respectively. Using the replay tool allowed us to start from the beginning of the development process and, step-by-step, simulate (i.e., "replay") changes made by engineers (e.g., modify the state of artifacts in Jira, add trace links, etc.) allowing us to automatically trigger constraint checks and track the process state the same way as in a "live" environment separately for each change. In other words, after each change event, we evaluated all process and QA constraints against the updated process snapshot.

To answer RQ1 for each constraint evaluation we evaluated (i) whether a step's Quality Check Document was fulfilled; (ii) which constraints were (not) fulfilled; and (iii) whether a step became active without the constraints of the predecessor step(s) being fulfilled.

To answer RQ2, we collected the following metrics for each process instance (i.e., a Jira artifact): number of Quality Check Documents un/fulfilled; number of un/fulfilled constraints; number of constraint checks performed; and the maximum number of past steps with unfulfilled constraints (i.e., how many steps an engineer advanced ahead without having the completion condition of the previous steps fulfilled).

To answer RQ3, we additionally captured the number of step cancelations, reactivations, and revocations only when replaying CS-3 as the process descriptions for CS-1 and CS-2 did not make use of exclusive (XOR) branching.

10. Results

10.1. RQ1: Process replay

Tables 3 and 4 report details regarding the QA constraint evaluation results across multiple process instances, grouped per process type for the three case studies.

AlwaysOk represents the number of process instances where engineers only progressed to subsequent steps when all quality constraints in previous steps were fulfilled. EventualOk reports the processes for which all constraints were eventually fulfilled. CompleteNotOk shows processes for which at least one constraint was never fulfilled. IncompleteOk counts those process
instances that were not finished by the end date of the time-frame but had all mandated constraints up to their current state fulfilled. IncompleteNotYetOk counts the partially completed process instances with unfulfilled constraints but no progress beyond those not fulfilled steps, in contrast to those with progress beyond that point as depicted in row IncompleteProgressedNotOk. Percentage values are reported relative to the sum of completed process instances, respectively sum of incomplete process instances.

**Dronology:** We noticed that for “Task” processes no completed process instance (i.e., finished “Task”) ever fulfilled every constraint before moving from one step to the next, yet around 30% fulfill all their constraints at the end, with ∼70% remaining unfulfilled at the end. “Sub-task” processes see ∼30% of instances “correctly” carried out, with only ∼50% not fulfilling their constraints. “Bug” processes are almost always correctly executed. “Design Definition” and “Hazard” processes are either correctly carried out from the beginning (the vast majority), or remain with unfulfilled constraints. When examining the incomplete process instances we encountered an expected large number of processes with unfulfilled constraints (i.e., hinting at steps with associated QA constraints that are not complete yet). However, we noticed that only a low percentage (<20% for IncompleteProgressedNotOk) of instances have engineers started too early on subsequent steps without having fulfilled the previous steps’ constraints.

**Frequentis:** For the case study residing in the safety–critical domain, we observed two interesting aspects. First, the number of SubWPs ultimately Ok reaches almost 90%, with the remaining 10% SubWPs showing unfulfilled constraints. To further investigate these, we manually examined the violating artifacts (exclusively LLReqs) and the comments attached to the SubWP Jira issue. Given that Jira is used as the primary means for communication and as coordination mechanism amongst the distributed teams and QA department, the comments provide an accurate and sufficiently complete track of the SubWPs history. For the 10 CompleteNotOk SubWPs, we found that in two cases SubWPs were used for documentation purposes rather than development, and therefore no trace links to Functional Units were present. In one case test cases were not applicable, and in three cases more than one Functional Unit was linked. This was due to the fact that the configuration subsystem affects multiple Functional Units. Three times a test case was referenced in the Jira comments (but no corresponding trace link in Jama was created). Once an additional SubWP was traced without closing the older one, and three times LLReq were marked for proposed future changes (and thus being no longer in state “released”). Note that some SubWPs experienced multiple, diverse violations. The second observation we made was that 11 SubWPs are IncompleteOk, even though could confirm that all the work was done. Manual investigation revealed that the Jira custom fields which are used by the passive process engine as a signal to advance the process were not used by the engineers, hence the process remained in the first step. We further discuss implications of these findings in Section 11.

**ACME-RA:** For all four projects we noticed that only a third or fewer of all process instances never incur a quality constraint violation. For P1 and P2 close to half or more of process instances remain with violations at their end. P3 and P4 fare much better in this respect with less than a third, or just 15% of completed process instance, respectively, having no constraint violations. The primary reason for unfulfilled constraints for completed process instances across projects P1, P2, and P3 was that the assignee for development was not a developer, but frequently assumed the role of team lead. For P4, the main reason was that not all child subtasks are closed or that a team lead or the assigned developer conducted the testing step. This violation temporarily also occurs in the other projects but is eventually resolved as all issues become closed at the end of the project. P4, however, was not completed at the time of data gathering and hence several process instances remained with an unfulfilled RA-C4 constraint. Addressing the incomplete process instances: P3 has three incomplete processes. Upon manual inspection of the issue’s change log, we noted that they missed the final transition from “resolved” to “completed”. P4 has 11 incomplete processes which all come with a violation of RA-C1, i.e., not having a developer assigned for development but the team lead.

<table>
<thead>
<tr>
<th>Table 4 Quality constraint evaluation results per process type (Frequentis and ACME-RA case studies).</th>
</tr>
</thead>
<tbody>
<tr>
<td>SubWP</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>AlwaysOk</td>
</tr>
<tr>
<td>EventualOk</td>
</tr>
<tr>
<td>CompleteNotOk</td>
</tr>
<tr>
<td>IncompleteNotYetOk</td>
</tr>
<tr>
<td>IncompleteProgressedNotOk</td>
</tr>
<tr>
<td>IncompleteOk</td>
</tr>
<tr>
<td>Total</td>
</tr>
</tbody>
</table>

**RQ1 Key Observation:** Temporary deviation from the prescribed process in the form of prematurely starting process steps without having preceding quality constraints fulfilled is common in open-source system development and industrial settings. The open source CS-1 sees most deviations for processes with multiple constraints, the industrial safety–critical CS-2 has most process instances executed in the expected step sequence even in the presence of multiple complicated constraints, while the industrial non-safety–critical CS-3 shows less deviation in processes that were part of more recent projects.

10.2. RQ2: Constraint fulfillment

Table 5 (for CS-1 Dronology and CS-2 Frequentis) and Table 6 (for CS-3 ACME-RA) displays for each constraint type and per each process type how often a constraint was fulfilled at the end of the process, and how often the constraint remained violated (i.e., an engineer did not fix it). In contrast to RQ1, where we observed the amount of fulfilled processes (and whether engineers deviated during the processes’ lifetime), here we obtain insights into which constraints are more likely to be fulfilled and hence are the root cause for a process to not fulfill all QA constraints.

**Dronology:** The left-hand side of Table 5 reports the differences in how often a constraint was fulfilled (limited to completed process instances). A majority of the constraints were fulfilled most of the time (~90% and higher). The lower fulfillment rates for constraints D-C1a and D-C1b (~<55%) are the main reason “Task” and “Sub-task” processes low in Table 5. EventualOk values in Table 3. Yet, constraints applied across multiple process types (i.e., D-C1a/b, D-C2, D-C3, D-C6, D-C7) exhibit similar fulfillment rates, i.e., a constraint is typically equally well fulfilled, respectively violated, regardless in which process type it is used.
**Frequentis:** For the second case study we could observe significantly higher fulfillment rates for all constraints (Table 5 right). The 12 unfulfilled constraint instances are distributed across the 10 Constraint SubWPs described above. Compared to the previous case study, a constraint for Dronology typically requires the existence of a trace link to one artifact (e.g., D-C1a: a Task traces to a least one Design Definition), whereas for Frequentis a constraint requires that all linked artifacts (i.e., LLReqs in ATC-C1 to ATC-C6) fulfill specific conditions. For ATC-C5, for example, a single LLReq out of 10 that does not have a trace link to a Functional Unit will cause the entire constraint to fail (regardless of whether all other LLReqs links are correct). To account for this, we further looked at the number of times an artifact (primarily an LLReq) was part of a constraint violation. With 1121 LLReqs and six constraints involving an LLReq, there are potentially 6726 opportunities that cause an overall constraint to fail. We observed 128, which is less than 2%. Out of 128 LLReqs that were part of a violation (due to missing, wrong, or superfluous trace links) only 3 were part of two different constraint violations. 98 LLReqs belonged to a single SubWP that was used for documentation (and needed no Functional Unit trace links), additional 12 LLReq belonged to a single SubWP where Test Cases were not applicable. The remaining 18 LLReqs violations were spread across the other eight CompleteNotOk SubWPs.

**ACME-RA:** In addition to the overall fulfillment rate, we also tracked whenever a step was completed (for the first time) but a constraint was still violated (χc).

We could observe that Constraint RA-C1 was the constraint that was violated most frequently upon process completion for three out of the four projects (P1, P2, P3). We further noticed that there was hardly any improvement, in terms of violations, during the process’ lifetime. Once a step was completed and exhibited a constraint violation, this violation was hardly ever repaired at a later stage. For these particular constraints, this is not necessarily surprising as the role of an issues’ assignee is determined by the skills (i.e., front end, back end, etc.) and thus unlikely to change even when reopened. Given the high intermediary and final fulfillment rate of P4, we hypothesize that the engineer at ACME-RA better recalled the most recent project (i.e., P4) compared to the older ones and that the appliability of a “Task” issue has changed over time.

In contrast, for the remaining constraints RA-C2, RA-C3, and RA-C4, we observe generally high final fulfillment across all four projects. They, nevertheless, differ in their intermediary fulfillment, with RA-C3 being almost always fulfilled in P1 to P3, i.e., tests were done by a QA engineer, reviews not being immediately done by a QA engineer (RA-C2), and not much attention is given to check whether all sub issues are indeed closed before closing the issue (RA-C4). As all issues are gradually closed over the project’s duration, so will this constraint be eventually fulfilled. The comparatively lower final fulfillment in P4 (compared to P1 to P3) was found to be due to the team lead or the assigned developer carrying out the tests instead of a QA engineer. After being prematurely started (i.e., having the predecessor step(s) incomplete). Note that we analyzed only completed process instances. Entries marked with a dash indicate that no process progress rules are in place, for example, to allow cancellation. Whether such rules can be in place depends on the underlying artifact details, here in particular the Jira issues states and their transitions. Hence, while only steps “Development” and “Testing” can be canceled, all steps can be reactivated, i.e., be marked completed but then experience additional engineering activities that violate the step’s postconditions.

For ACME-RE we observed that engineers, for “Task” issues, made heavy use of the flexibility offered by the Jira state transitions. This flexibility is consequently also supported by the ProCon framework. We see every step that can be canceled was indeed temporarily canceled in every project (except for “Testing” in P1). Unsurprisingly, there is some correlation with the fulfillment of constraints. P1 and P2 see more cases of a non-developer assigned to the development step, while P3 and P4 fulfill this QA constraint more often, hence the latter two projects see fewer cases of premature starting the “Development” step. Premature execution of the “Closing” step is particularly high in P2, which is explained by the lack of either (No)Reviewing or (No)Testing steps being carried out (note the sum of (No)Reviewing and (No)Testing only amounting to 13 and 11, respectively, over a total of 21 process instances). Here, the engineers typically transitioned the “Task” issue into state “Closing” directly from “In Development”. This is another case, where the valid transitions in Jira changed over time, but nevertheless, our framework was able to track this deviation. Inspecting the number of step revocations (i.e., implying repeated execution of the same step), we note that developing and testing are especially often repeated in P4. This could be an indicator that engineers aimed to provide more fine-granular feedback of their task progress by switching often between the respective Jira issues states. In contrast, reactivation primarily occurs for the “Preparation” step (across all projects), as this step does not come with preconditions that are invalidated (and hence no revocation is observable but rather only reactivation).

**RQ2 Key Observation:** Quality assurance constraint violations are common but to a lesser extent in industrial safety–critical environments, where QA constraints are mandated by regulation. In the open source CS-1 violations are mostly caused by forgotten or incorrect traces, while in the industrial safety–critical CS-2 violations are mostly caused by process edge cases. Constraint violations in CS-3 are primarily caused by the issue’s assignee not having the expected engineering role.

10.3. RQ3: Process flexibility

ACME-RA’s process offers more flexibility to select a step compared to the processes from Dronology and Frequentis. Table 8 shows the number of steps that were started (i.e., became ACTIVE, are COMPLETED) and percentage of these steps being (temporarily) canceled, reactivated (i.e., having reached a completion state and then becoming active again), revoked (i.e., being active and then having the preconditions violated), and, finally, being prematurely started (i.e., having the predecessor step(s) incomplete). Not show that we analyzed completed process instances. Entries marked with a dash indicate that no process progress rules are in place, for example, to allow cancellation. Whether such rules can be in place depends on the underlying artifact details, here in particular the Jira issues states and their transitions. Hence, while only steps “Development” and “Testing” can be canceled, all steps can be reactivated, i.e., be marked completed but then experience additional engineering activities that violate the step’s postconditions.

For ACME-RE we observed that engineers, for “Task” issues, made heavy use of the flexibility offered by the Jira state transitions. This flexibility is consequently also supported by the ProCon framework. We see every step that can be canceled was indeed temporarily canceled in every project (except for “Testing” in P1). Unsurprisingly, there is some correlation with the fulfillment of constraints. P1 and P2 see more cases of a non-developer assigned to the development step, while P3 and P4 fulfill this QA constraint more often, hence the latter two projects see fewer cases of premature starting the “Development” step. Premature execution of the “Closing” step is particularly high in P2, which is explained by the lack of either (No)Reviewing or (No)Testing steps being carried out (note the sum of (No)Reviewing and (No)Testing only amounting to 13 and 11, respectively, over a total of 21 process instances). Here, the engineers typically transitioned the “Task” issue into state “Closing” directly from “In Development”. This is another case, where the valid transitions in Jira changed over time, but nevertheless, our framework was able to track this deviation. Inspecting the number of step revocations (i.e., implying repeated execution of the same step), we note that developing and testing are especially often repeated in P4. This could be an indicator that engineers aimed to provide more fine-granular feedback of their task progress by switching often between the respective Jira issues states. In contrast, reactivation primarily occurs for the “Preparation” step (across all projects), as this step does not come with preconditions that are invalidated (and hence no revocation is observable but rather only reactivation).

**RQ3 Key Observation:** ProCon is able to accurately describe the process progress even in the presence of frequent repetition, pausing, or skipping of engineering activities. This is especially relevant in highly iterative development processes (e.g., CS-3 P4) that frequently switch between development and testing phases.

10.4. RQ4: Performance of the ProCon framework

Constraints related to certain Quality Check Documents are typically validated by engineers to ensure all QA demands for their step are fulfilled. Overall, the replay of 26,926 change events over 109 simultaneously active process instances from the Frequentis case study resulted in 18,241 Quality Check Document evaluations. The resulting replay of events from the Tool connector’s cache including constraint evaluation took ~6.5 mins (averaged over 10 evaluation runs). This corresponds to ~0.02 s necessary for evaluating all quality constraints within a single Quality Check Document: a duration that allows frequent and timely feedback to developers.

Similarly, we evaluated the performance for the ACME-RA case study which exhibits simpler quality constraints, but more
complex constraints for process progress tracking. Here we replayed almost 52,500 change events, out of which almost 10,500 were relevant for at least one process instance and thus resulted in a trigger of ProCon. On average, evaluating the 267 process instances took \( \sim 2.1 \) mins.

**RQ4 Key Observation:** With an average individual constraint evaluation duration of \( \sim 0.02 \) s ProCon is able to quickly evaluate artifact changes and subsequently provide timely feedback on quality assurance constraints and process status even in the presence of many simultaneously active process instances. Rather, the main factor influencing timely feedback to developers is the rate at which artifact updates are made available to ProCon and how frequent developers visit the process dashboard.

### 10.5. RQ5: Ease of use

To evaluate how easily constraints can be created using ProCon, we conducted a preliminary user study to gain initial insights in the way ProCon is used and what challenges users face (c.f Section 9.1). Five out of six participants successfully encoded all six StepTransitionsRules. Results are more varied for the DataMapping rules (see Table 7). We found similar result diversity for QA constraint writing.

All participants stated that they found writing the constraints intuitive, with those not finishing them explicitly stating that they felt confident to complete them if given more time. Two of the six participants, however, found the DataMapping rules a bit confusing. One participant stating “while the tasks themselves are easy, it is hard to enter the mindset.” and another participant commenting “having multiple rules for one step was confusing”. Specifically for the DataMapping rules, we hypothesize that they are more difficult to grasp as these constitute not true/false evaluations but require adding and removing of artifacts from the step’s output.

### 11. Discussion

The analysis of the data collected from the first case study, Dronology, indicated that the actual process – in some cases significantly – deviated from the planned one. Upon requesting feedback, a project lead at Dronology explained that while guidelines and a development process were in place, it was not always feasible to follow them by the letter. Student teams were involved...
in the development of some of the components, and while they
have been trained on the process, they still lacked experience
in following all prescribed rules and guidelines. Furthermore,
besides the software development aspect, the focus was also on
obtaining a data set of trace links, and that the process had to be
adapted to the availability of open-source developers. Rather than
forcing a change of process which might be infeasible, the insights
gained here could be used to decide where to introduce additional
QA checks, e.g., making constraint check results available upon re-
viewing a pull request. Here ProCon would then highlight where
traces are missing or are incorrectly set. A trace recommendation
technique such as Rath et al. (2018) and Antoniol et al. (2002)
could further assist in establishing the trace itself.

In contrast, the analysis of Frequentis’ SubWPs confirmed that
engineers do in fact follow the stringent quality standards one
would expect in the (highly safety-critical) ATC domain. The
finding that 10% of process instances EventuallyOK confirms the
QA engineer’s experience that engineers need support for pro-
ducing correct and complete trace links as significant additional
work at a later stage was necessary. Our investigations of the
CompleteNotOk instances highlighted that, on the one hand,
corrections come with significant coordination effort, and still
may result in missing traces or incorrectly set artifact properties.
On the other hand, the investigations highlighted the presence of
edge cases where the QA constraints do not apply, reinforcing
the need for sometimes tolerating these inconsistencies. ProCon
offers two options in such a case: first to ignore the constraint
evaluation results, and/or to adapt the process, respectively con-
straints. In either of these two cases, a rigid (thus inflexible),
active process enforcement environment would have severely
hampered the engineer’s available actions, effectively forcing the
engineer to work outside the defined process. Finally, the huge
amount of >18.000 Quality Check Document evaluations explains
why manually providing timely feedback is infeasible.

Finally, for the third case study, ACME-RA’s software product
has no stringent safety implications, and no external regulation
that enforces the use of strict quality standards. The use of pre-
scribed engineering processes is motivated mainly internally to
obtain an accurate picture of the overall development progress
within the individual projects. We observed simpler constraints,
and lower degrees of following these constraints. Here ProCon
is helpful in obtaining a true picture of the progress, e.g., by
highlighting those “Task” processes that are officially closed, but
still have child tasks that are not closed yet, hence indicating that
there is still work to be done. Tracking processes and constraints
violations also supports engineers in inspecting how often certain
steps are indeed found to be applicable, and when executing
these occasional steps, whether they are indeed executed as
planned (i.e., are quality constraints indeed fulfilled). Over the
course of time, process engineers may then learn whether the
fulfillment rate of QA increases, and the rate of unfixed QA
violations goes down. For the ACME-RA case study, our ProCon
also demonstrates the ability to track repetition and deviation
of process steps and the frequent encounter of such behavior, a
strong signal that temporary deviations are pervasive in software
engineering efforts. Table 9 summarizes the research questions
outcome per applicable case study.

11.1. Implications for practitioners and researchers

Based on the observations made from the case studies, we can
conclude that ProCon can have significant practical implications
for QA engineers. Supported by automated checks for “standard”
cases, they can shift attention and focus on edge cases and de-
viations from the process. Furthermore, they can allocate time
for improving constraints checks, and investigating whether these
checks and following the process actually result in better software
quality (Conradi et al., 1994a). Engineers can leverage the imme-
diate feedback they receive on their work status and do not need
to revisit their work at a later, inconvenient time. The various
stakeholders no longer need to build their own (error-prone)
custom “helper tools” that are almost infeasible to maintain or to reuse across multiple projects or teams.

We received very positive responses from engineers at Frequentis upon presenting ProCon with one team lead wishing to have it ready as a product by tomorrow, and a QA engineer joking to be out of work then. While the prototype was applied only to one product group at Frequentis, we are currently rolling out the prototype to three more product groups, each having different rules (but use Jira/Jama), thus only the process and rules need to be adapted. Given the excellent performance during replay (i.e., handling 27k artifact changes across 109 process instances within a few minutes) we are confident that adding more rules in the current rollout will not lead to performance problems. We subsequently expect to obtain more detailed insights into the prototype’s practical use.

Beyond the immediate practical applicability, we would expect that our approach leads to cost savings by reusing constraints due to treating constraints and their evaluation results as first-class citizens (which also reduces maintenance costs). Constraints may be modified over time to accommodate changes in the organization’s process, or may apply in diverse process contexts, making them amenable to product line engineering approaches. Further, the concepts of software product line engineering could be used to manage the variations in process and QA concerns found in a larger organization (Simmonds et al., 2015). We would also expect cost savings by reusing constraints across different systems subject to the same regulation. As a regulatory standard is applicable for a wider range of systems, constraints (and also processes) could be formalized at the level of the standard for reuse by affected companies. We are however aware, that initially reuse would occur primarily within an organization for two main reasons. First, constraints are rather tool-specific, especially how an organization makes use of their tools’ extensibility through custom fields and custom link types. (We could imagine a mapping from a more generic/high-level standard-centric constraint to an organization’s low-level tool-specific constraint; the feasibility of this is largely unclear). Second, from our discussions with various regulated companies, we understand that many see their precise processes and constraints for implementing a regulatory standard as confidential. We, hence, expect reuse at the level of a regulatory standard to find traction once processes and/or constraints are no longer considered a competitive advantage but rather an opportunity for jointly reducing development costs.

With respect to implication for researchers, ProCon has further potential to serve as a platform for additional research prototypes and support tools built on top of it. Passive process execution has the benefit of enabling inspection at any time to what degree the process is followed and where deviations have occurred (respectively are not mitigated yet). Deviations can thus be detected earlier, e.g., an engineer has started too early on a step. Alerts or mitigating actions may then be less invasive rather than significant rework later on. Other potential support mechanisms could build the engineer in how to set up the correct output artifacts, or direct the engineer in how to fix a constraint violation or offers to automatically fix it Mayr-Dorn et al. (2021).

Our approach can serve as the basis for other research on supporting the software engineering process such as proactively driving the process through automated actions. Here, open points for investigation include: how the process context can be used to better drive CI/CD pipelines, and automatically prepare engineering artifacts and engineering activities such as reviews. In general, the questions that emerge from automation also need to focus on the negative sides, such as engineers trusting too much in tool support or choosing guidance actions that are the most convenient for them but perhaps not optimal for the overall development process.

12. Threats to validity

Internal Validity. We address researcher bias by modeling process and constraints from an open-source system and two companies rather than conducting controlled experiments. ProCon works on arbitrary artifacts, traces, and change events and was not specifically tailored to Jama or Jira.

External Validity. Based on the limited scope of our evaluation with two different systems, we cannot claim generalizability of our findings. However, we argue in line with Briand et al. (2017) that context-driven research will yield more realistic results. Our work evaluated the ability of ProCon to passively execute diverse engineering processes and QA constraints (simple ones from an open-source system, as well as medium and more complex ones from industry) in a timely manner. We analyzed data from these three sources with two being “production data” from an industrial safety-critical system and an industrial non-safety-critical system. Typically, being able to obtain such data, and furthermore being able to publicly report results is quite challenging as companies are reluctant to provide insights into their working processes at that level of detail, and open-source systems rarely come with such extensive explicit artifacts and trace information.

Construct Validity. For RQ1 and RQ2 we addressed the question of how frequently process deviations occur and which specific constraints are violated, by replaying real historical data. Hence, we stepped through the process as it occurred with exactly the same sequence of changes and evaluated the process and QA constraints. We thus measure the “official” state of the process as it would be used as evidence to demonstrate compliance with regulations. We cannot, in this way, measure the tacit process state implicit in the minds of engineers. Engineers might have used informal communication channels to convey status information while forgetting/delaying the update the intended process status signals (e.g., Jira issues status or checkboxes). We thus might generate a more pessimistic view of the process state. As the purpose of ProCon is to provide guidance to engineers and evidence for compliance, however, we believe it is important to minimize the gap between measured and tacit process states by highlighting deviations.

For RQ3, for assessing whether ProCon support flexible processes, we measured if for a process with alternative steps and engineers frequently repeating steps, we indeed find the reactivation, cancellation, and revocation of steps as indicated in the replayed artifacts’ history. For RQ4, we measured only the performance of the core process and QA constraints evaluation as this aspect is computationally expensive. Poll frequency for obtaining artifact updates might have an effect on the responsiveness as rare polling with the subsequent potentially large amount of updates could lead to longer constraint evaluation times. This aspect is also determined by the network load, tool load, and artifact update frequency which is different in each deployment scenario and hence needs to be assessed on a use-case basis.

For RQ5 we conducted a controlled experiment to evaluate ease of defining realistic constraints. Our aim was to obtain insights into whether the initial learning curve is sufficiently low to promote uptake by practitioners. Long-term use in a production environment needs to be separately investigated as actual constraint complexity might vary in practice. To mitigate any threats we provided information to participants about the concepts we were investigating, communicated the purpose of the study to our participants, carefully discussed the study setup and execution among multiple researchers, and conducted a pilot study.

12.1. Limitations

The evaluation process is exemplary of the processes at Frequentis, but does not cover all of ED109. The model and engine
however are not specific to ED109 and can be adopted to the specific process setting as shown with Dronology and ACME-RA that followed a completely different process and TIM. Adopting a different scenario then is mostly a matter of connecting different tools. Contemporary tools tend to come with a HTTP/REST interface, or client implementation (as did Jira and Jama with dedicated Java clients). Hence, it requires little effort in wrapping these clients for integration with the engine. New tools (and artifacts) are then accessible in the rules.

We also make the assumption that step completion can be detected from tools. The need for management, team leads, and project leaders to obtain an accurate picture of progress, as well as having teams increasingly work distributed across multiple locations leads to a move away from informal signaling of completion toward explicit one, e.g., assigning a different member to an issue, setting a checkbox, setting the status of an issue, etc. Thus we believe that obtaining such indicators in almost all cases is reasonable.

Note, that overall, we cannot make any claims on the completeness of our approach as our research was primarily guided by the needs of our industry collaborators and the attempt to avoid investigating irrelevant aspects, the equivalent of the software development pitfall YAGNI: “you ain’t going to need it”. As we continue to apply our approach and framework to additional scenarios, we expect to identify missing aspects, especially along the lines outlined here (Mayr-Dorn et al., 2021). However, we want to be able to validate the solutions to those aspects under realistic settings which, in the context of this work, typically requires an industry evaluation partner.

13. Conclusions and outlook

In this paper, we presented an approach for reducing the effort of ensuring that development activities adhere to quality constraints. The novel aspects are the decoupling of QA constraints from process control and dataflow, which allows engineers to deviate from the process when necessary, whilst informing them which constraints are yet unfulfilled and which steps are already complete. Our framework achieves this flexibility by merely observing the engineers’ actions in their tools rather than restricting them in their allowed activities. Constraints get constantly reevaluated upon changes even for steps that should not be worked on yet or which have been already marked as complete. Our evaluation using both, industry and OSS projects, revealed that engineers frequently deviated from the intended process for some time and that our approach can identify missing (traceability) information required by regulations.

Future work focuses on two main aspects. First, we intend to study the effect of having our prototype in use by engineers at Frequentis. This will include design, prototyping, and evaluation of advanced features, such as actionable guidance suggestions on how to return to the prescribed process upon deviation and change impact notification across steps. We aim to quantify the actual effort reduction and gather qualitative feedback for further improvements. Second, we will study QA engineers and process engineers during the creation, evolution, and maintenance of process models (including constraints) with ProCon to understand how their task can be supported even better. This will include experimenting with constraint code completion techniques, automatically creating premature start conditions, and constraint deadlock checking techniques. Ultimately, we aim to quantify the cost savings by comparing the effort to specify and maintain processes to the benefits of easier and less error-prone collection of QA evidence.

CRediT authorship contribution statement

Christoph Mayr-Dorn: Conceptualization, Software, Investigation, Data curating, Writing – original draft, Validation. Michael Vierhauser: Conceptualization, Investigation, Data curating, Writing – original draft, Validation. Stefan Bichler: Software. Felix Keplinger: Software, Data curating. Jane Cleland-Huang: Conceptualization, Supervision, Writing – review & editing. Alexander Egyed: Supervision, Writing – review & editing. Thomas Mehoffer: Validation, Data curating.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The introduction contains a link to a figshare folder.

Acknowledgments

This work was funded by the Austrian Science Fund (FWF) under the grant numbers P31989, P29415-NBL, and P 34805-N, and by the state of Upper Austria through LIT-2019-8-SEE-17 and LIT–2019–7–INC–316 as well as the LIT Secure and Correct Systems Lab. This work has been also supported by the FFG, Contract No. 881844: “ProFuture”. The Dronology case study was supported by the United States National Science Foundation under grants SHF:1741781 and CPS:1931962.

References


Christoph Mayr-Dorn is a senior researcher at the Institute for Software Systems Engineering at the Johannes Kepler University Linz, Austria. He holds a Ph.D. in Computer Science from the Technical University Vienna. His current research interests include software process monitoring and mining, change impact assessment, and software engineering of cyber–physical production systems.

Michael Vierhauser is a senior researcher at the LIT Secure and Correct Systems Lab at the Johannes Kepler University Linz, Austria. He holds a Master’s degree in Software Engineering and Ph.D. in Computer Science from the Johannes Kepler University Linz. His current research interests include Cyber–Physical Systems, Safety Assurance, and Runtime Monitoring.

Stefan Bichler Finished his Master Degree in Software Engineering at the Johannes Kepler University Linz, Austria.

Felix Keplinger Finished his Master Degree in Software Engineering at the Johannes Kepler University Linz, Austria.

Jane Cleland-Huang is a Professor and Chair in the Department of Computer Science and Engineering at the University of Notre Dame. Her research interests focus upon Software and Systems Traceability for Safety–Critical Systems with a particular emphasis on the application of machine learning techniques to solve large-scale software and requirements engineering problems. She is the lead PI of the DroneResponse project, Chair of the IFIP 2.9 Working Group on Requirements Engineering, and Associate Editor of the Communications of the ACM.

Alexander Egyed is a Full Professor and Chair for Software-Intensive Systems at the Johannes Kepler University, Austria (JKU). He received a Doctorate degree from the University of Southern California, USA in 2000 and then worked for industry for many years before joining the University College London, UK in 2007 and JKU in 2008. He is most recognized for his work on software and systems design—particularly on variability, consistency, and traceability.

Thomas Mehofer is head of ATC Communications at Frequentis AG.
5.3 Using Constraint Mining to Analyze Software Development Processes

Authors:
Thomas Krismayer, Christoph Mayr-Dorn, Johann Tuder, Rick Rabiser, Paul Grünbacher

Venue:

DOI:
https://doi.org/10.1109/ICSSP.2019.00021

Abstract:
Most software development organizations nowadays use issue-tracking tools to manage software processes throughout the life-cycle. Still, understanding development processes, keeping track of process execution, and reacting to deviations in projects remains challenging. In particular, the actual process usually differs from the process perceived by developers, making it hard to define the processes developers are expected to carry out. This is further challenged by frequently changing processes and process variations in different projects and teams. In this paper we describe an empirical study in which we applied a constraint mining approach from the field of software monitoring to automatically extract process definitions in the form of constraints. Specifically, we applied the approach to datasets extracted from four real-world projects (using the Jira issue-tracking tool) in a company developing a recreational activities platform. The mined constraints describe the boundaries of the actual processes and thus help to understand process behavior. Constraints can be frequently re-mined to understand process evolution. The mined constraints can also be used to monitor future processes to detect problems in the development process early on. We involved a domain expert to evaluate the usefulness of our results and investigated to what extent the mined constraints reflect the official development process of the company. We also report mining results for different issue types, across projects, and over different time windows.
Using Constraint Mining to Analyze Software Development Processes

Thomas Krismayer*, Christoph Mayr-Dorn†§, Johann Tuder*, Rick Rabiser*, Paul Grünbacher†§

*Christian Doppler Laboratory MEVSS, †Institute for Software Systems Engineering
Johannes Kepler University Linz, Austria
§Pro2Future GmbH, Linz, Austria

{thomas.krismayer, christoph.mayr-dorn, rick.rabiser, paul.gruenbacher}@jku.at

Abstract—Most software development organizations nowadays use issue-tracking tools to manage software processes throughout the life-cycle. Still, understanding development processes, keeping track of process execution, and reacting to deviations in projects remains challenging. In particular, the actual process usually differs from the process perceived by developers, making it hard to define the processes developers are expected to carry out. This is further challenged by frequently changing processes and process variations in different projects and teams. In this paper we describe an empirical study in which we applied a constraint mining approach from the field of software monitoring to automatically extract process definitions in the form of constraints. Specifically, we applied the approach to datasets extracted from four real-world projects (using the Jira issue-tracking tool) in a company developing a recreational activities platform. The mined constraints describe the boundaries of the actual processes and thus help to understand process behavior. Constraints can be frequently re-mined to understand process evolution. The mined constraints can also be used to monitor future processes to detect problems in the development process early on. We involved a domain expert to evaluate the usefulness of our results and investigated to what extent the mined constraints reflect the official development process of the company. We also report mining results for different issue types, across projects, and over different time windows.

Index Terms—Process mining, constraint mining, process evolution.

I. INTRODUCTION

It is difficult in large software projects to keep track of the progress of the many individual tasks performed, e.g., developing new features, fixing bugs, or refactoring the system. Most software development organizations nowadays thus use issue-tracking systems, which offer process guidance and support for process monitoring via basic tracking capabilities. However, detecting tasks that take unusually long or deviate in other ways from the expected process still remains difficult.

When defining, analyzing, and improving software processes it has turned out useful to distinguish several views [1]:

- the perceived process (what developers think they do), i.e., their subjective perspective on the process based on their personal and daily work experience;
- the actual process (what developers really do), i.e., how they execute the process, e.g., as determined through observation; and
- the official process (what developers are supposed to do), i.e., the process model defined by an organization.

Obviously, these views will commonly differ in practice. For instance, processes are frequently changed due to current project needs. Further, they typically vary in different projects and teams of an organization. Thus, when improving software development processes, understanding the perceived, actual and official processes provides the foundation for defining an initial process, which can be continuously improved towards a target process based on experience and observation [1].

In this context, observing processes is useful to detect deviations from the official processes early, thus giving teams the opportunity to react accordingly. Currently, process monitoring is often based on the official process available before the start of a project (e.g., as defined in an issue-tracking tool). Since the actual behavior in a project may differ significantly, using the official process as a reference for comparison is often not feasible. Process mining approaches [2]–[4] have thus been proposed to automatically learn constraints characterizing the actual process, which can then be used to monitor and improve development.

Following this direction, we describe the application of a constraint mining approach [5], [6] developed in our earlier research on software monitoring [7] to analyze software development processes. We first extract data from an issue-tracking tool and then reveal process definitions in the form of constraints. Specifically, our approach can extract different types of constraints on event occurrence, timing, and data (as well as combinations of these). The approach presents candidate constraints to users in a domain-specific language [8] and offers a range of filtering and ranking strategies [6] to select the constraints to be used for monitoring.

In the application context described in this paper, the mined constraints are based on actual real-world process data of bug fixes, system improvements, or feature development. For these processes (issues) we analyze their creation times, state changes (e.g., created, to be verified, or resolved), and the people (roles) involved. Examples of mined constraints include typical sequences of state changes for a particular issue type, the duration to close an issue, or the developer role changing the status of a bug, e.g., to resolved. Such constraints allow monitoring the actual states and behavior of processes to detect development problems early. Constraints can also be frequently re-mined to understand process evolution by highlighting process changes.
Specifically, this paper provides the following contributions based on empirical investigations: (1) We use a dataset extracted from the Jira issue-tracking tool\(^1\) from four real-world software development projects of a company developing a recreational activities platform to evaluate the usefulness of our approach and to assess the quality of the mined constraints. (2) We also investigate to what extent the constraints cover the official and the actual processes, and discuss the impact of different issue types, projects, and time windows in projects on the results.

In the remainder of this paper we first present a motivating example (Section II) and briefly describe our constraint mining approach (Section III). We then describe our research approach (Section IV) and report qualitative feedback we received from an industry expert (Section V). We present quantitative analyses on the impact of different issue types, projects, and time windows in projects on the mined constraints (Section VI). Finally, we discuss our results (Section VII) and related work (Section VIII). We conclude the paper with an outlook on future work.

II. MOTIVATING EXAMPLE

The issue-tracking tool Jira allows developers to define processes by defining different issue types for common development scenarios, e.g., development or bug-fixing tasks. For each issue type the modeler can define the available states and the allowed transitions. For instance, Figure 1 shows an excerpt of the allowed process states and transitions for the issue type Task. Once a new issue is created its initial state is Open. The state changes to In Development as soon as the work on the task starts. After finishing the task the developer changes the state to Ready for Review. A tester then picks up the task and assesses the issue before changing the state to Reviewed. If a task is regarded as finished, its state changes to Resolved (one of the allowed end states), either after review or immediately after development. If required, testing can be performed after resolving the task (state In Testing). Both development and testing can be suspended (states Suspended Development and Suspended Test), e.g., if additional resources are required, and then resumed when these resources become available.

Real-world data sets contain many different issue types comprising numerous states and possible transitions. In our data set, for instance, the issue type Task provides 24 different states. In development projects, issue types are instantiated often hundreds of times, making it very difficult to keep track of all currently enacted processes. As a result, many common cases of unintended behavior cannot be detected by the existing capabilities of issue-tracking systems, as the following three examples show: In case of delays a difficult development task may remain in state In Development for an unusually long time. Role violations occur if a state is changed by an unauthorized user, e.g., if a task that is Ready for Review is not Reviewed by a tester. Finally, combined violations may happen: for instance, moving an issue from Suspended Test back to In Testing has to be done within a reasonable time to avoid delays caused by blocking the testing task. While Jira enforces process tracking via state transitions it does not support transition conditions, hence any user with write access may update the issue state without time limitations within the bounds of the defined transitions. Defining and monitoring process constraints, e.g., on delays and role violations, thus becomes desirable.

However, manually defining constraints is infeasible in practice due to the high number of projects, issue types, and instantiated issues in organizations. Additionally, constraints will need to vary for different types of issues (e.g., a bug fix will usually require less time than implementing a new feature) as well as different projects and teams (e.g., some projects may have a separate quality assurance (QA) group, while in others the developers are responsible for testing too). Overall, this suggests the use of process mining to automatically reveal actual process characteristics on the fly.

III. BACKGROUND: CONSTRAINT MINING

Our example shows the need for an approach to automatically extract process constraints, which reflect the actual process behaviour. We use a constraint mining approach [5] that can discover different types of runtime constraints from recorded events and event data items. It finds temporal constraints checking if multiple events occur in a specific order, value constraints checking data items attached to events, and hybrid constraints combining temporal checks and value checks. For example, in our motivating example the temporal constraint “if event In Development occurs event Ready for Review occurs within 30 days” can detect unexpected delays of issues remaining in state In Development for more

---

\(^1\)http://www.atlassian.com/jira
than 30 days. The value constraint "if event Reviewed occurs data(UserType) = "QA"" checks for role violations. The constraint is violated, if a user with a role other than quality assurance changes the state of a ticket to Reviewed. Finally, the constraint "if event Suspended Test occurs event In Testing where data("UserType") = "QA" occurs within 21 days" is an example of a hybrid constraint. The constraint is violated if a ticket stays in Suspended Test for more than 21 days or if, e.g., a developer changes the state of a ticket from Suspended Test to In Testing.

Our existing constraint mining approach [5] takes event logs recorded from system executions as input and automatically suggests such constraints. The event logs can be created based on the change history of issue-tracking tools as we will describe in Section IV. Our approach performs the following steps:

Step 0: Transform the events to a uniform event representation. As our constraint mining approach is supposed to work with different systems, it is necessary to transform the input data to a common format, which allows easy parsing of the events, their timestamps, types, and attached event data. For example, changing the asignee of an issue in Jira leads to an event of type Assignee with event data User Type, i.e., the role of the new assignee. A transition to state In Progress (e.g., from state Open) is interpreted as an event of type In Progress, again with event data on the person triggering the transition.

Step 1: Detect event sequences. In the first mining step our algorithm automatically extracts event sequence types, i.e., multiple events frequently occurring together in a specific order. The algorithm first identifies all pairs of events that occur in 95% of all process executions (in our context: Jira issues), for example, <Open, Assignee>, or <Open, In Development>. Starting with one of these pairs, the algorithm then forms longer event chains by repeatedly extending it with additional pairs with one overlapping event type. The two example sequences could be combined into two possible longer event sequence types: <Open, Assignee, In Development> and <Open, In Development, Assignee>. The algorithm discards event sequence types with insufficient support and selects the sequence with the shortest overall duration. Next, it separately collects all event sequence instances that follow an event sequence type, for example, all events from Jira issues where Open is followed by Assignee before the next Open event. If an event sequence type occurs multiple times within an issue, the algorithm produces distinct event sequence instances for each occurrence. In the first step our algorithm produces temporal constraints by determining the interval between the first and the last event (per event sequence type) that includes 98% of all sequence instances. The list of event sequence instances for each sequence type is then further processed in the next step.

Step 2: Create feature vectors. Our algorithm next generates a feature vector for each sequence instance extracted in step 1. The data fields available in the events of a sequence instance determine the vector’s content. For example, for sequences of type <Open, Assignee>, the vector may contain <Open.byUserRole, Open.byUserId, Assignee.byUserRole, Assignee.byUserId, Assignee.byUserRole, Assignee.byUserId>. Our algorithm uses these vectors to mine value constraints by checking field values that remain unchanged for all sequences – mining them as constants and removing them from the vectors to prevent their use in the next step.

Step 3: Analyze feature vectors. When analyzing the created feature vectors our algorithm aims to find specific patterns, such as multiple values remaining unchanged across different events (e.g., Open.byUserId equals In Development.byUserId), values that are equal for a majority of all sequences (e.g., Open.byUserType equals “Developer”), and intervals that contain at least 98% observed (numeric) values for one field. Our algorithm mines these cases as hybrid constraints.

Step 4: Filter, group and rank constraint candidates. Finally, all extracted constraints are first filtered, e.g., to remove duplicate or very similar constraints, then grouped into lists of similar constraints, and finally ranked, e.g., depending on how often they evaluate to true for the input dataset.

IV. RESEARCH APPROACH

We empirically evaluate our constraint mining approach using data from real-world software development projects. Specifically, we analyze event logs created from Jira logs to uncover constraints of development processes that need to hold during process enactment. We use datasets from four projects of an industry partner. The company is in the business of hosting a recreational activities web platform. The company’s identity and project names have to remain confidential due to the sensitive nature of the analyzed data. We obtained data from four projects: P1, a low-priority Android app development project; P2 and P3, two business-critical Android app development projects; and project P4 integrating two types of recreational activities that involved experts beyond front-end, business logic, design, and database engineering (e.g., marketing and legal departments).

We investigate the following research questions:

- RQ1. Are the constraints mined from the process observations useful?
- RQ2. How do the mined constraints differ with respect to different types of processes (issue types)?
- RQ3. How do the constraints mined for one issue type differ across multiple projects?
- RQ4. How do the mined constraints change in different time windows within the projects?

Specifically, our research approach shown in Figure 2 comprises the following steps:

1. Preparation of Datasets. We first extracted process observation data from Jira. We then performed a user/role mapping based on the ids of the users. To this end, we applied a manually compiled dictionary of user ids to their roles. Example roles were Quality Assurance, Database Engineer, and Web Development. We then separated the datasets to allow investigating different issue types (RQ2), projects (RQ3), and time windows (RQ4).
(2) Creation of Event Logs. We pre-processed the Jira datasets to create event logs containing events (of different types) with timestamps and event data as input for our constraint mining approach. Our parser interpreted all changes to issue fields (i.e., assignee, state, fix version, etc.) in the Jira datasets as events. Each event belongs to one particular issue and contains the time when the issue was changed as well as the role of the person creating or changing the issue. As described above, issues have different types – such as task, bug, or improvement. We use this information to define the types of the events (task-related event, improvement-related event, etc.). The four projects P1–P4 contain a total of 1017, 2676, 1052, and 939 issues, respectively. During our experiments we only used the five most common issue types (Task, Bug, Improvement, Localization, and Project Management), reducing the number of issues to 832 (10524 events), 2459 (30207 events), 967 (10475 events), and 818 (10262 events).

(3) Constraint Mining. We then mined constraints for the actual process of each project to detect event sequences and their duration, state transition rules, feature vectors of the detected sequences, and patterns in the contained field values. For each of the four projects our approach produced a list of constraint candidates. We removed constraint candidates with very low support, that were evaluated to true less than ten times when checked on the complete event log.

(4) Assessment of Usefulness (RQ1). To address our first research question we asked a domain expert to analyze the constraint candidates for the largest project (P2) to qualitatively assess the usefulness of our constraint mining approach. Specifically, we mined constraints for the five most common issue types (Task, Bug, Improvement, Localization, and Project Management) of P2. These five issue types contain between 971 and 236 individual issues – the sixth largest issue type of this project contains only 84 individual issues. As a preparatory step, we presented the states (but not the transitions) of each issue type to the domain expert and asked him to explain the states and all possible transitions. This helped to prepare the mindset of the expert (cf. our discussion of perceived vs. actual processes). We then presented the list of mined constraints and asked the expert to assess the usefulness of each constraint candidate using the following options: yes (i.e., useful without further adjustments), yes with minor changes (i.e., useful after adjusting, for example, the duration of a sequence), or no (i.e., not useful). We considered all constraints rated with yes and yes with minor changes as useful, because they can eventually be used for process definition and process monitoring.

We also asked the domain expert to assess the criticality of all useful constraints as high, medium, or low. The criticality of a constraint indicates the severity of violating it: low-criticality constraints may yield warnings, while high-criticality constraints may indicate crucial problems or disregard of the defined processes.

Finally, the domain expert answered open questions to provide additional feedback and comments. Specifically, we asked if he was missing important constraints, whether the mined constraints triggered any ideas for additional constraints, how surprising he regarded the found constraints, and how well the automatic constraint grouping [6] works.

(5) Analysis Across Issue Types (RQ2). We also compared the constraints mined for different issue types. Specifically, we analyzed differences in the mined constraints for issues of type Task, Bug, Improvement, Localization, and Project Management in project P2. For the mining, we used the same settings as for the qualitative evaluation in the previous step (RQ1). Before mining, we split the datasets (and the related event logs) into multiple folds, i.e., each containing issues (and related events) of one specific type only. We mined constraints for each fold individually and then compared them.

We considered constraints as equal if they were identical; if only their duration differed (e.g., the same sequence may be restricted to eight days for one issue type but to ten days in another case); or if only the value of the checked data differed (e.g., the role closing an issue).

(6) Analysis across Projects (RQ3). We additionally compared the mined constraints across projects. We again mined constraints for the five most common issue types Task, Bug, Improvement, Localization, and Project Management, this time from all four projects. The four projects contain between 150 and 1000 issues of type Task and Bug, respectively. For the other three issue types (Improvement, Localization, and Project Management) we left out one project containing less than 15 issues.

We mined a list of constraint candidates for each of the projects and then compared these lists to find equal constraint candidates. Same as for the analysis across issue types (see previous step), we also considered constraints as equal if they checked the same sequence and event data items, but differed regarding duration or checked data values.

(7) Analysis of Constraint Evolution (RQ4). Finally, we repeatedly mined constraints on different time windows through-
out the four projects to analyze changes in the found process constraints. Specifically, we split the issues from one issue type into multiple time windows, mined constraints individually for each of these time windows and for all four projects, and analyzed the evolution of the resulting constraints over time.

Due to the setup of this final experiment we reduced the number of evaluations a constraint had to pass in order to not be filtered out from ten to five, thus compensating the lower number of issues remaining in the different time windows.

V. Results of the Qualitative Evaluation (RQ1)

Table I contains some examples of mined constraints. For the largest project (P2) 127 constraints were found, between 15 and 34 for the different issue types. Following our research method we presented these constraints to the domain expert, a key developer of the project, to evaluate their usefulness. Table II summarizes the feedback. Overall, the domain expert regarded 69.3% of all constraints as useful. The ratio of useful constraints is above 50% for all issue types. 18 of the 20 constraints that were rated as yes with minor changes just need a minor adjustment: they contain one of the two user roles assigned to developers and have to be modified by adding also the other developer role, as commented by the expert. Constraints rated as not useful are, e.g., user type checks for states that are also allowed to be performed by other users.

The assessments of the domain expert regarding criticality can be seen in Table III. In total 88 constraint candidates were regarded as useful by the domain expert. The expert rated the criticality of 47 of these constraints as high, 18 as medium, and 23 as low.

Finally, we asked the domain expert five open-text questions complementing the quantitative evaluation of the constraint mining results. The domain expert reported that all important constraints he could think of were contained in the list of mined candidates. Looking at the mined constraints triggered the expert to think about possible process improvements, e.g., merging some states (such as In Development and In Development Again) in Jira. He also gave two examples for constraints he found surprising at first but understood after a second look. The domain expert also rated the grouping of constraints [6] our approach provides as meaningful.

VI. Results of the Quantitative Evaluation

Following our research approach, we split the dataset into multiple folds, each containing issues from one specific issue type. We mined constraints for each fold individually and compared the found constraints. We built groups of equal constraints from different folds and assessed differences between these groups and between constraints that could not be matched as they were valid only for certain folds.

A. Analysis Across Issue Types (RQ2)

In the first experiment we compared the constraints found for the five most common issue types of the largest project P2. A total of 46 constraints only appeared for one issue type, nine constraints were mined for two different issue types, nine for three issue types, and also nine for four issue types. No constraint was found for all five issue types.

The number of constraints appearing for each pair of issue types can be found in Table IV – the numbers along the main diagonal represent the total number of constraints mined for a particular issue type. Pairs of issue types can be read horizontally and vertically in the table: for example, horizontally we notice that 18 of the 24 constraints mined for issue type Improvement (i.e., 75%) are also mined for issue type Bug. Vertically, 18 out of 25 constraints for type Bug are also mined for type Improvement (i.e., 72%). In the following we report only the direction with the higher number. The types Bug and Improvement share the largest number of common constraints, followed by the pairs Improvement-Task and Bug-Task, for which more than half of the constraints found for the first type were also found for the second one (66.7%,
Table IV
NUMBER OF CONSTRAINTS SHARED AMONG ISSUE TYPES (RQ2).

<table>
<thead>
<tr>
<th></th>
<th>Task</th>
<th>Bug</th>
<th>Improvement</th>
<th>Localization</th>
<th>Project Mgmt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task</td>
<td>34</td>
<td>14</td>
<td>16</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>Bug</td>
<td>14</td>
<td>25</td>
<td>18</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>Improvement</td>
<td>16</td>
<td>18</td>
<td>24</td>
<td>3</td>
<td>11</td>
</tr>
<tr>
<td>Localization</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>15</td>
<td>2</td>
</tr>
<tr>
<td>Project Mgmt</td>
<td>12</td>
<td>9</td>
<td>11</td>
<td>2</td>
<td>29</td>
</tr>
</tbody>
</table>

respectively 56%). For example, constraint #3 from Table I was mined for these three issue types, but not for Localization or Project Management. Project Management shares most constraints (41.4%) with Task, while Localization has most constraints in common with type Bug (26.7%).

24 out of the 46 constraints found for only one issue type are triggered by states only defined for this particular type, and are thus irrelevant for any other type. An example is constraint #5 in Table I, which can only be mined for Project Management, because no other issue type uses the state Requirements Review. Ten out of the 46 constraints are triggered by states that occurred less than ten times for each other issue type and would thus be filtered out. The remaining 12 constraints are triggered by events that occurred at least ten times in at least one other issue type but did not result in any constraints – nine of these twelve constraints were considered useful by the domain expert.

This first experiment demonstrates that many of the mined constraints are overlapping at least between some of the issue types (such as Task, Bug, and Improvement). Two examples are constraints #1 and #2 in Table I that are mined for all issue types except for Localization – with a different duration for each issue type (e.g., #1 with 50 days and #2 with 5 days for type Bug).

There are also issue types (such as Localization) with constraints checking distinct states, thus resulting in a smaller number of mined constraints with relevance for other issue types. This could be expected when considering the underlying process: tasks, bugs, and improvements all regard development activities, contrary to the the more specific issue types Localization and Project Management.

B. Analysis across projects (RQ3)

The results from the previous experiment showed that the different issue types lead to diverse constraints (e.g., using different states). Thus, to address our third research question (analysis across projects), we compared only the constraints mined for the same issue type across the different projects. Constraint candidates for the issue types Task and Bug were mined from all four projects, while for the issue types Improvement, Localization, and Project Management only three of the four projects contained a sufficient number of issues. The number of projects each of the constraints appears for can be seen in Figure 3.

A large share (83.3%; 35/42) of the constraints mined for issue type Project Management was found only for one project, such as constraint #5 in Table I. 19 out of these 35 constraints are triggered by events that appeared less than ten times for other projects, thus not passing the defined threshold. This shows the diversity of Project Management issues across the projects, which led to many different constraints.

For the other four issue types Task (44.6%), Bug (42.5%), Improvement (54.5%), and Localization (55.5%) the share of constraints only mined for one project is significantly lower. One constraint example mined for issue type Task for all projects is constraint #1 in Table I; its duration ranges from 27 days (P4) to 84 days (P3).

The differences for issue type Project Management become even more obvious when taking a more detailed look at the number of constraints mined for each issue type across the four projects (cf. Figure 4). The number of constraints mined for issue type Project Management varies between five for P3 and 29 for P2, while this span is smaller for all other types.

The results show differences between the projects for all five issue types. This is most obvious for issues of type Project Management, where only 16.7% of all distinct constraints are mined for more than one project. For the other issue types, however, up to 57.5% (for issue type Bug) of all distinct constraints could be mined for at least two different projects. These numbers suggest that the different projects follow their
C. Analysis of Constraint Evolution (RQ4)

For the final experiment, to address RQ4, we compared constraints from different time windows that are related with the same issue type. We matched equal constraints and counted for each distinct constraint the number of time windows it appeared in. Similar to the previous experiment, we did not mine constraints for issue types with less than 15 issues (Project Management for P1, Improvement for P3, and Localization for P4). Hence we analyzed 17 data series, one for each combination of project and issue type as depicted in Figure 5.

Figure 5 specifically shows how many constraints appear in how many time windows. In 15 out of 17 cases the highest number of constraints appear only in a single time window. For seven of the 17 cases, the second highest number of constraints appear in all time windows. For the types Bug in P2 and Task in P3, the constraints appearing in all five windows even yield the highest count. Overall, the distribution of short-lived constraints (that live for only one or two time windows) and constraints that can be generalized to several other time windows (or even for the entire project) is very different for the four projects.

An example mined consistently for all time windows in P2 is constraint #2 in Table I. The duration for the individual windows ranges from 8 to 23 days. Constraint #4, however, is only mined for the first window. This happens, because...
the trigger event Ready to Implement is only observed in the beginning of the project.

Figure 6 shows the number of constraints that were mined for each issue type and time window for the largest project P2. While the absolute number of constraints mined exclusively in one time window is high, the relative number for the issue types Task, Bug, Improvement, and Localization is only between 10% and 18%. One exception regards issue type Project Management, for which 31% of all mined constraints cannot be matched to a constraint mined from another time window. Additionally, 20 of the 52 distinct constraints for this issue type are mined for two time windows, while only five are mined for three or more windows (cf. Figure 5). This finding is similar to the results from the previous research question that also suggested that constraints for Project Management can often not be generalized.

These results indicate that while some constraints are only valid for a short period of time, the majority is also valid for other project stages or even the entire project. However, we also showed that the distribution of generalizable constraints can vary heavily between different projects and issue types. This suggests frequent re-mining to understand process evolution.

VII. DISCUSSION AND THREATS TO VALIDITY

Regarding RQ1 – are the constraints mined from the process observations useful? – we reported very positive feedback from our industry expert who considered 88/127 of the mined constraints useful, rated many (47) as highly critical, and made several other positive comments.

Regarding RQ2 – how do the mined constraints differ with respect to the observed issue types (types of processes)? – our analyses show that many constraints are similar for development-related activities (tasks, bugs, and improvements), while highly-specific processes such as localization lead to very specific constraints. As could be expected, these results show the difficulty of defining (and subsequently monitoring) one-size-fits-all processes, thus confirming the importance of automatic mining.

The experiments we performed for RQ3 – do the mined constraints differ across multiple development projects? – demonstrated that for all issue types, differences can be found in the constraints mined for different projects. Most differences could be observed in project management activities, while we noticed fewer differences in development-related activities (task, bug, improvement). Overall, similar constraints could be mined even across multiple projects for processes having similar characteristics.

With respect to RQ4 – how do constraints evolve over time? – the experiments showed that there are both constraints that are only mined for a short period of time and constraints that are mined for multiple (or even for all) time windows in a project. The detailed results from one of the projects showed that for most issue types constraints are mined for multiple time windows, which suggests that they can remain useful for other stages of the project. This also suggests frequent re-mining of constraints to support process evolution.

As any empirical study, our experiments exhibit a number of threats to validity:

Internal Validity. We address researcher bias by analyzing data from an actual company rather than conducting controlled experiments. The mining algorithm works on arbitrary events and was not specifically tailored to Jira issues or the used datasets.

External Validity. Rather than claiming for wide generalizability of our results, we argue in line with Briand et al. that context-driven research will yield more realistic results. In this paper, we thus evaluated the usefulness of the mined constraints for a company with an engineer working at that company (i.e., the domain expert). We analyzed data from a single company only as such data from real-world, industrial environments is extremely hard to get. Companies are very reluctant to provide insights into their working processes at that level of detail. As different engineers and roles (e.g., database expert, designer, team lead) have different concerns, they might evaluate the constraints’ usefulness differently. The positive feedback from an engineer familiar with the development processes under study, however, at least shows that the approach is applicable and indeed useful in the observed context.

We make no claim that our approach will yield equally useful results when applied to data from a Jira server used for open-source development. Often these servers, such as hosted by the Apache Software Foundation provide only the default issue types and issue states and thus require developers to limit their coordination to processes based on this limited set of states, or apply other non-structured mechanisms such as comments, mailing-lists, and tacit knowledge to manage processes. Even less structured are issues in projects hosted on GitHub. GitHub issues are either “Open” or “Closed”, any intermediary state needs managing via arbitrary labels (i.e., tagging) with no support for defining or restricting valid transitions.

Engineers assigned to departments and roles are a second, context-specific characteristic of the analyzed data set. Jira itself is unaware of roles (i.e., who should be changing an issue’s state) and hence such information cannot easily be extracted from logs. As many of the constraints mined from the data set include roles, we cannot infer how useful our approach will be for environments where no role information is available or where roles and departments are not clearly assignable. This will often be the case in open-source projects because they rarely exhibit a clear department and/or role assignment structure. Hence a comparison of the four projects to open source projects would make little sense.

VIII. RELATED WORK

We describe related work in the software process mining community, which focuses on algorithms, techniques, and

1https://issues.apache.org/jira
2https://help.github.com/articles/about-issues/
frameworks to identify constraints among process steps as well as discover process models from process execution traces. These traces most often include commit information from source code version repositories, but also other implicit process support tools such as email lists and bug trackers. Please note that we keep aside work on constraint mining in the business process management field as the subject of a business process is typically an independent document, request, or case, but not a set of interrelated source code artifacts, which exhibit quite diverse characteristics, hence findings cannot be reliably transferred.

The need for process mining and approaches such as the one introduced in this paper emerges from software engineering’s nature to follow a process. Explicit processes, however, assume control of the process over tools and engineers, which is greatly limiting freedom, respectively forces engineers to work outside the process to handle unforeseen situations and optimizations not foreseen by the process. Diebold and Scherr [10] show that in industrial practice the majority of processes, therefore, focus on description rather than using formal notations or models. Organizations tend to apply semiformal process descriptions containing different graphical, table-based, or structured-text elements for representation.

Hence, explicit process support focuses, on the one hand, on engineering parts that can be automated such as in continuous deployment and continuous integration [11] or dependency tracking between software design and code artifacts [12]. On the other hand, when involving human tasks, approaches aim for integrating process information into tools (e.g., [13]) or focus on the single developer micro-level. For example, Zhao et al. apply Little-JIL for describing fine-grained steps involved in refactoring [14] or help developers track artifact dependencies during rework [15].

Process constraint mining thus becomes a necessity to obtain information on how much the actual process really matches the official process, highlights process improvement potential, and serves as input to run-time constraint monitoring, which provides rapid feedback to process stakeholders.

Poncin et al. [2], [16] introduce the Framework for Analyzing Software Repositories (FRASR) for combining data from source code repositories, email lists, and bug trackers. They subsequently utilize the ProM process mining framework for obtaining insights such as classifying developers in open source software projects to roles such as project leader, core member, peripheral developer, bug fixer, or reader. They also analyzed the typical transitions between bug report states on Bugzilla. The main difference to our approach is that with FRASR/ProM, the engineer has to define what type of constraints/relations to look for and what the relevant properties of the observations are. Rubin et al. [3] suggest to extract software engineering processes from source code repository information. Their approach relies on classifying artifacts into meaningful categories, e.g., README, CONFIG, or SRC. Maggi et al. [17] propose to mine declarative process models from events logs. Their DECLARE approach produces a set of LTL constraints. Gupta et al. [18] conducted process mining across an issue-tracking system, a code review system, and a version control system. They map events from these systems into a single process (based on states) and determine transition occurrences. Based on this annotated transition diagram, they analyze the bug-fixing process from reporting to resolution to discover bottlenecks, deviations from the intended process, joint activities, and work handover. Similarly, Akbarinasaji et al. [19] mine a bug report’s lifecycle for predicting bug-fixing duration. Bala et al. [20], [21] propose an approach for mining GANTT charts from source code commit history. It requires extensive, explicit mapping of commits to activities.

Early work on software process mining applied diverse techniques such as mining a petri-net from versioning logs [22], generating a finite state machine [23], using probabilistic relational modeling [24], or probabilistic event analysis [25]. All these approaches assume that events (e.g., from a commit) adhere to a well-defined, a-priori-known set of activities. Later approaches focusing on mining continuously evolving processes similarly rely on explicit activity types [26]–[28].

Our approach distinguishes itself from the state-of-the-art software process mining primarily by the fact that it does not require data labeling such as classifying artifacts into meaningful categories and/or mapping events to activity types. Hence, the entry barrier to applying our mining algorithm is very low. It focuses on frequently occurring event sequences, including value constraints as well as duration. This greatly supports the exploratory investigation of how the actual processes manifest as it increases the chances of detecting dependency types process designers didn’t anticipate.

IX. CONCLUSION

In this work we have reported how constraint mining can be used successfully to identify constraints describing software development processes. Mining can be used to analyze similarities and differences between different issue types and projects and to analyze the evolution of development processes. A domain expert of our industry partner considered our approach as very useful to mine (critical) constraints describing their development processes. The mined constraints could be used for continuous monitoring to detect deviations of the actual process from the expected and the official process. Additionally, they could be used to give feedback to process owners and for process modeling.

In future work, we plan to extend the presented approach to investigate such applications in detail. Furthermore, we intend to apply the approach to additional datasets from further software development projects, also from other domains. Also, it would be interesting to compare the results obtained with our approach to constraints discovered by other software process mining approaches.

ACKNOWLEDGEMENTS

The financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for
Research, Technology and Development, and Primetals Technologies is gratefully acknowledged. Christoph Mayr-Dorn would also like to acknowledge partial funding by the Austria Science Fund (FWF): P29415-NBL funded by the Government of Upper Austria as well as FFG, Contract No. 854184. Pro²Future is funded within the Austrian COMET Program-Competence Centers for Excellent Technologies – under the auspices of the Austrian Federal Ministry of Transport, Innovation and Technology, the Austrian Federal Ministry for Digital and Economic Affairs and of the Provinces of Upper Austria and Styria. COMET is managed by the Austrian Research Promotion Agency FFG.

REFERENCES

5.4 Process Inspection Support: an Industrial Case Study

Authors:
Christoph Mayr-Dorn, Johann Tuder, Alexander Egyed

Venue:
14th ACM International Conference on Software and System Processes ICSSP (2020)

DOI:
https://doi.org/10.1145/3379177.3388900

Abstract:
Organizational factors such as team structure, coordination among engineers, or processes have a significant impact on software quality and development progress. Projects often take much longer to complete than planned and miscommunications among engineers are common. Yet, the process for exploring the project-specific or organization-specific root causes why this happens is still poorly supported. Investigations are cumbersome and require significant effort. In the context of this industrial case study, our industry partner was interested in measuring and assessing how the organization structure and issue handling processes ultimately affected software quality and time. Reducing the effort of such investigations/retrospectives and speeding up fact finding is important as it allows for more frequent, informed engineering process improvements and feedback to managers, team leads, and engineers. This paper describes our approach of pairing process metrics with visual historical inspection of issues. Stakeholders such as managers, team leads, or quality assurance engineers inspect metrics (and deviations from expected values) for individual issues and utilize a historical visualization of the affected (and related) issues to obtain insights into the reason for the metric (deviation) and its root cause. We demonstrate the usefulness of our approach based on our ProcessInspector prototype providing access to data on four real industry projects and a qualitative evaluation with team leads and group leads from our industry partner.
Process Inspection Support: an Industrial Case Study

Christoph Mayr-Dorn
Institute for Software Systems Engineering
Johannes Kepler University, Linz Austria
christoph.mayr-dorn@jku.at

Johann Tuder
formerly ACME
Linz, Austria

Alexander Egyed
Institute for Software Systems Engineering
Johannes Kepler University, Linz Austria
alexander.egyed@jku.at

ABSTRACT
Organizational factors such as team structure, coordination among engineers, or processes have a significant impact on software quality and development progress. Projects often take much longer to complete than planned and miscommunications among engineers are common. Yet, the process for exploring the project-specific or organization-specific root causes why this happens is still poorly supported. Investigations are cumbersome and require significant effort. In the context of this industrial case study, our industry partner was interested in measuring and assessing how the organization structure and issue handling processes ultimately affected software quality and time. Reducing the effort of such investigations/retrospectives and speeding up fact finding is important as it allows for more frequent, informed engineering process improvements and feedback to managers, team leads, and engineers. This paper describes our approach of pairing process metrics with visual historical inspection of issues. Stakeholders such as managers, team leads, or quality assurance engineers inspect metrics (and deviations from expected values) for individual issues and utilize a historical visualization of the affected (and related) issues to obtain insights into the reason for the metric (deviation) and its root cause. We demonstrate the usefulness of our approach based on our ProcessInspector prototype providing access to data on four real industry projects and a qualitative evaluation with team leads and group leads from our industry partner.

CCS CONCEPTS
• Software and its engineering → Software creation and management.

KEYWORDS
issue, organizational structure, software engineering process metrics, prototype, JIRA, history visualization

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM Reference Format:

1 INTRODUCTION
Software engineering projects often last much longer than planned and miscommunication between engineers occurs on a regular basis. The process for exploring the exact project- or organization-specific root causes why this happens is still poorly supported. Investigations are cumbersome and require significant manual effort. Retrospective analyses, such as done at the end of sprints in agile development environments, are important to happen jointly as a group effort but benefit significantly from tools that assist in better understanding the cause of undesirable situations such as missed milestones. Reducing the effort of such investigations/retrospectives and speeding up fact finding is important as it allows for more frequent, informed engineering process improvements and feedback to managers, team leads, and engineers.

Research over the past decades has shown that organizational factors such as team structure, coordination among engineers, or processes have a significant impact on software quality and development progress [16]. Engineers working on strongly coupled artifacts tend to require frequent communication to coordinate their engineering efforts [22]. Hence, achieving socio-technical congruence (STC) is one aspect towards improving software development performance [6].

Therefore, in the context of this industrial case study investigated in this paper, our industry partner was interested in measuring and assessing how its organizational structures and issue handling processes ultimately affect coordination among engineers and timely delivery in order to obtain insights in how and where to focus on improvements. Typically, software process metrics provide such insights and multiple research efforts aim to improve them [7, 8, 15, 20]. Yet, determining which metrics are useful and accurately describe the ongoing development efforts is non-trivial as this differs among companies and often also among departments and groups within the same company.

To this end, we propose an approach of pairing process metrics with visual historical inspection of issues to overcome the limitations of metric inspection without context on the one hand and visualization without guidance on the other hand. Stakeholders such as managers, team leads, or quality assurance engineers inspect metrics (and deviations from expected values) for individual issues and utilize a historical visualization of the affected (and related) issues to obtain insights into the reason for the metric (deviation) and its root cause. We designed the accompanying prototypical tool (ProcessInspector) light-weight and flexible to support easy integration and adaptation of metrics. We demonstrate the usefulness of
our approach based on a prototype providing access to data on four real industry projects and a qualitative evaluation with team leads and group leaders from our industry partner.

The contributions of this paper are:

- A flexible approach for combining process metrics and issue history visualization
- A prototypical tool implementation (Process Inspector)
- A data set describing the complete set of issues from four industry projects
- A qualitative evaluation of the approach

The remainder of this paper is structured as follows: Section 2 provides case study background information. We introduce our approach in Section 3 and the corresponding prototype in Section 4. We qualitatively evaluate and discuss our approach in Section 5. Section 6 compares our approach to related work, before Section 7 concludes this paper with an outlook on future work.

2 CASE STUDY BACKGROUND

2.1 Industry Context

ACME is in the business of hosting a recreational activities web platform. The company’s identity and project names have to remain confidential due to the sensitive nature of the analyzed data. At the time of data extraction and paper evaluation, the company was structured into ten departments. Those departments consisted of 22 groups. The following groups are of primary interest for this paper: software developers-frontend, software developers-backend, database, graphics, quality assurance, and product & project management. These groups are heavily involved in the software development process. The departments are physically distributed across two buildings. A project team usually consists of at least one member from each of the above listed groups of interest. When a project’s software is released for public access the project team remains responsible for maintenance. This implies that the team works together during the product’s complete life-cycle from initial project setup, to implementation, and ongoing support even though team members are situated in different offices. This cross-departmental/group organization style comes with a significant number of necessary meetings and informal communication channels within the team. From experience, this often resulted in miscommunication, unclear assignments, and in undocumented decisions in any of the used tools: the company uses JIRA, Confluence, Skype, rocketChat and Outlook to keep track of projects. JIRA is used as ticketing system and also provides further information such as comments and lists of changed source code files. The intended process to follow for software development is reflected in JIRA issues, issue relations, and issue properties such as state, milestones, releases, and assignee.

2.2 An Industry Challenge

As most software development companies, ACME is interested in improving its workflows and software development process. After each completed major project the current process is evaluated and adapted to fit new organizational circumstances such as new teams. To this end it needs to compare differences among teams at project and at issue level (e.g., issues bottle necks at a specific group) to establish how differences in process and structure affect performance. ACME also needs to distinguish if problems result from e.g. workflow flaws, inefficient organizational structure, or insufficiently accurate artifacts such as unclear requirements. Given ACME’s organizational structure, a particularly interesting question was whether vertically organized teams (team members from different groups become co-located) perform better than horizontally organized teams (team members from different groups remain with their groups). This evaluation process is cumbersome currently as the reasons for deviations and comparisons across teams and projects are not easily obtainable from ACME’s tool landscape.

2.3 Process Improvement Method

One typical approach to process improvement is through a Goal-Question-Metric (GQM) driven method [5]. In the context of our industry partner, the purpose of such a study is to evaluate the impact of the current team structure and issue ticket usage on efficiency and coordination effort from the point of view of team-leads (manage projects) and group-leads (manage an expertized centered set of engineers such as testers) in the context of lightly distributed teams.

According to the taxonomy by Smite et al. [21], ACME’s teams can be classified as Location: Onshore, Legal Entity: Insourcing, Geographic Distance: Close, Temporal Distance: Similar. These teams can nevertheless be considered distributed as already a separation by floors or buildings can significantly reduce informal contacts and thus influence coordination [1].

ACME identified four key questions that they need to answer on their path to process improvement:2

1. Is engineering effort accurately estimated?
2. How much coordination is happening?
3. How efficient are coordination actions?
4. How efficient is project planning?

Here questions and metrics (see Section 3.1) are iteratively refined upon feedback from our industry partner.

---

1 We identify a member of such groups as an engineer and use the term developer when explicitly referring to a member from the frontend or backend group.

2 Note, that these are not the research questions to be answered in this paper.

---

Figure 1: Approach to assisting process inspection in the scope of a GQM method: full/black arrows show input/output between GQM steps, dashed/blue arrows depict supported feedback between steps applying the Process Inspector prototype.
3 APPROACH

Our approach (see Figure 1) supports the stakeholders during metric interpretation by pairing metric calculation (and presentation) with issue event timelines as metric context. As a side effect, our approach assists the stakeholders to determine which metrics are useful and applicable in measuring process improvement (in our case study with a focus on the subgoal of measuring team coordination).

The GQM method (as applied by ACME) typically consists of following six steps:

1. Define improvement goals.
2. Develop questions that allow the goals to be quantified.
3. Determine which metrics answer these questions.
4. Implement data collection mechanisms.
5. Collect and interpret the metrics for immediate feedback/ improvement.
6. Assess how the gathered metrics support reaching the goal and derive recommendations.

Deriving accurate and meaningful metrics is difficult as their measurement context changes over time (e.g., teams are restructured, processes change) or they become applied in situations they are not sufficiently suitable for (low priority, short term, technology exploration project). As a consequence, metrics need careful interpretation in their measurement context.

Our approach addresses this concern by supporting a stakeholder to easily move between metric calculation, respectively inspection, (here on the level of issues) and the context that gave rise to that particular metric instance (here the history of the issue and its related issues).

In our particular case, this allows a stakeholder to become aware (at a coarse-grained level) of projects that seem to be poorly performing, and (at a fine-grained level) of issues that need attention. The stakeholder then inspects the visual history of these issues that give raise to a critical metric value to understand if there is truly a problem at hand, whether the particular situation constitutes an exception to the rule, or whether the metric is not suitable (anymore) in the larger context of this project. For example, metrics that measure how long an issue is in state Open and In Progress before it is Closed become unreliable when engineers forget to transition an issue upon starting their work and only briefly set the issue to In Progress before closing it. This type of deviation from expected behavior results from the flexibility engineers require. Too rigid, explicit process control greatly limits engineers’ freedom, respectively forces engineers to work outside the process to handle unforeseen situations and optimizations not foreseen by the process. Diebold and Scherr [10] show that in industrial practice the majority of processes, therefore, focus on description rather than using formal notations or models. Visual inspection provides, on the one hand, rapid feedback on the brittleness of metrics and, on the other hand, points out potential for tool improvement to assist engineers in following the intended process.

Likewise, the timeline visualization allows stakeholders to browse issue progress and, upon finding suspicious looking event sequences, the stakeholder can easily cross-check with metrics how that particular issue measures against other (similar) issues.

In short, our approach, respectively prototype, directly addresses step 4 and 5 of the GQM method, and indirectly supports the re-evaluation of metric selection in step 3.

Ultimately, the research question addressed in this paper (and answered in the evaluation section) is: Is the combination of the provided process metrics and issue history visualization effective for obtaining insights into coordination problems? Note that it’s not a goal of this paper to investigate whether the proposed metrics or the GQM method indeed lead to process improvement.

3.1 Metrics

We selected the following set of metrics and refined them together with engineers at ACME to ensure they are relevant in ACME’s development context. The metrics come at various granularity levels: per-issue, per-occurrence (multiple times per issue possible), informational/aggregating (information at issue level, project-level metric value).

1. Issue resolved/closed date <= issue due date: The resolved date is set e.g., if an engineer finishes a bug-fix implementation. The closed date is set when e.g., a tester finished testing of a bug-fix. In the ideal case every issue should be closed before its due-date. For this metric the relevant fields of an issue are the closed-, resolved- and due-date. A negative difference value describes an issue that was closed/resolved before its due-date, a positive value that it missed its due-date deadline, respectively. Large deviations in either direction are indicators of estimation errors. Positive deviations indicate that engineers required less time, negative deviations indicate unforeseen complexity in implementation or coordination. (per-issue metric: duration in days)

2. Due dates aren’t changed: Due dates are set at the beginning of a project. Project managers and engineers then arrange the workload and determine when the various components of a project should be completed. Due date changes indicate that project planning was not accurate (e.g., lower/higher effort/coordination estimation) or an assigned engineer (no longer) needed to work on a higher prioritized task. Specifically, this metric counts the number of due-date changes per issue (not considering the initial setting of the due-date, which is also a change event) and derives a project overall ratio. (per-occurrence metric: duration in days for from/to due date change; project-level metric: ratio of “issues with changes” compared to “issues without changes”)

3. Fix version wasn’t changed: A fix version identifies the release or milestone this issue’s result should be available in. Fix version changes reflect issues being moved between milestones during planning or implementation (or even when already completed). Many such changes may indicate challenges in project planning and also the release workflow. This metric is calculated similar to M2 but based on changes to the fix version property. (per-occurrence metric: changes: count of from/to fix version changes; project-level metric: ratio of “issues with changes” compared to “issues without changes”)

4. Duration in approve state: Some issues have to be approved by team leads before work can start. Usually such an approval process does not take very long and this metrics informs the
responsible decision maker when delays happen repeatedly. Approval delays block the actual engineers from working on the issue. This metric is calculated only for issues that have had a status changed to “Approval necessary”. The temporal distance to the next status entry is then the metric value.

(5) Re-open distance to due date: Issues that are re-opened multiple times might indicate an engineer-to-issue assignment mismatch or unclear/incomplete requirements. Whenever an issues state is changed to “Open”, “Open Again” or “Re-opened” the change date’s difference to the due date is calculated. A positive value describes re-opening after the due date, a negative value a re-opened before the due date. (per-issue metric: how often changed; per-occurrence metric: how long before due date in days)

(6) Assignee Changes: Assignee changes are part of the workflow. For example, a project manager creates an issue and assigns it to an engineer. After the implementation the issue is assigned to a tester. A high number of changes potentially indicates communication problems between departments or that an issue was not implemented correctly. Especially assignments within a department, e.g., from front-end developer to another front-end developer, should not happen. Therefore the metric is split into the number of intra-department changes and the number of cross-department changes. (per-issue metric: number of changes)

(7) Use of comments: ACME expects complex issues to require additional refinement and clarification via comments, especially when an issue involves engineers from multiple departments. This metric subset also provides an indication whether discussions, decisions, and assumptions are documented within Jira or using other tools. (per-issue metric: comment count, commenter count, count per commenter, min/max/avg comment length).

(8) Issue re-assignment without documented communication: Changing the assignee typically requires no documentation when the engineers follow the intended workflow, e.g., a developer finishes the implementation and a tester has to start. However if an issue re-assignment happens multiple times and deviates from the usual workflow then this should be documented in the comments. The metric uses the changelog entries of type “assignee”. The metric counts for each issue, how many pairs of such changes exist without a comment in between. For each of these change pairs, the metric also collects the duration between assignment changes in days. (per-issue metric: number of assignments, total duration in days between two assignments without comment)

(9) Issue re-assignments without status changes: Usually an issue is re-assigned to another engineer if the issue’s status changes, e.g., the developer finishes implementation and sets the status to “Resolved”, assigns it to a tester who changes the status to “In Testing”. If issues are re-assigned without a status change something could be wrong either in the workflow or how engineers handle issues. This metric uses the same assignee changelog entries as the previous metric M8 but considers status changelog entries instead of comments.

(10) Duration between re-assignment and subsequent status change: Issue re-assignment indicate that another engineer can start their work, indicating this start by updating the issue’s state (see previous metric). This metric measures for each re-assignment the duration until the subsequent status change. A long timespan could indicate that the assigned engineer is overloaded and that the issue should have been better worked on by someone else, respectively that project planning overlooked/created a bottleneck. (per-issue metric: duration in days)

(11) Issue resolved date compared to code freeze date: The metric shows the difference between the resolve date of an issue and the code freeze date. Usually issues should be resolved before code freeze because during code freeze no new features are implemented for the current milestone and the time to release is reserved for bug fixing and testing. Issues resolved after code freeze indicate work overload and wrong project planning or normal bug fixing. A positive value indicates that an issue was resolved before the code freeze, a negative value the opposite. (per-issue metric: difference in days to code freeze date)

(12) Issue of future milestone started in an earlier milestone: The metric calculates the time span between the first progress of an issue to the start date of its milestone’s predecessor. This metric highlights which and how many issues of a project were started earlier than planned. This indicates that some engineers do not have enough issues assigned or that project planning was not accurate enough: towards the end of the milestone, engineers were already doing work for another milestone and they could have moved more features into the current one. A positive time span shows that an issue was started before the official milestone start, a negative number shows the opposite. (per-issue metric: difference in days to milestone)

All these metrics rely on a small set of issue change event (i.e., timeline events). Displaying them in their temporal order as they have occurred (sometimes in isolation, somethings almost simultaneously) allows to better place the metric values in the engineering process context.

3.2 Issue Timeline Events

The selected issue timeline events are driven by the metrics that make use of these events. As such, these events are exemplary and we make no claim for completeness (additional/different metrics may introduce other events). These events, however, are central to coordinating engineering efforts. In general, we distinguish between two event categories: events that are placed at the time they occurred and events that represent predefined dates such as fix-versions and due dates. The currently considered event list comprises:

(1) created: marks the date the issue was added to the issue tracker (here Jira).

(2) status: marks changes to the status, e.g., from “Open” to “In Progress”.

84
(3) assignee: marks handing over the ticket from one responsible team member to another.
(4) fix version set: marks changes to the release version in which this issue should be included.
(5) resolved: marks when the issue was set to resolved.
(6) due date: marks when the issue is planned to be completed.
(7) due date changes: marks updates to the due date.
(8) comment: marks when an engineer commented on the issue.
(9) last updated: marks the last change to any of the issue properties.

Figure 2: Prototype Architecture.

4 PROTOTYPE TOOL SUPPORT
In this section we present the current prototypical tool support for calculating metrics and inspecting issue histories. We deliberately aimed for a maximally light-weight prototype to allow for rapid prototyping and iterative refinement.

The current prototype (see Figure 2) consists of three main components: the Jira Extractor, a Database for caching processed issues, and a Web Frontend for calculating and displaying metrics and issue history.

The Jira Extractor is responsible for accessing a Jira server’s REST API for retrieving all issues of a given project as json documents (in our industry partner’s case identified by a root issue with all related issues obtained via an issue’s “Part” relations). It then reads an engineer’s department from a configuration file, anonymizes the engineer id, prepends the department id to the anonymized engineer id, and replaces this throughout the issue json document before storing each issue json document in the CouchDB.

CouchDB is a schemaless JSON database merely used to store the processed issues and making them available to the Web Frontend. The Web Frontend is a locally hosted HTML page making heavy use of javascript libraries for calculating and displaying metric result tables, timeline visualization, and connecting (directly) to the CouchDB. The set of metrics can be easily adapted as each metric is implemented as a separate class and located in its respective, separate javascript file (e.g., metric 1 in M1.js). Each metric class simply needs to provide a calculate, getDataTable, and getSummaryDataTable method for triggering metrics calculation across all issues of the project, returning the overall metric result table with values for each issue (for per-issue metrics) or occurrence (for per-occurrence metrics), and a summary table displaying, for example, averages, maximum, and minimum values (customizable per metric), respectively. A standard workstation is sufficiently fast to carry out issue retrieval (from the CouchDB), metric computation, and visualization on demand without the need to include server side data (pre-)processing facilities. The Web Frontend consists of the following UI elements (see also Figure 3):

- **Project Selector** (A, top left corner) is a dropdown input that allows the user to pick a project. Each project is stored in a separate “database” in CouchDB.
- **Timeline Scope** (B, left) enables the user to select start and end dates for the timeline. The timelines for all visible issues shrinks or expands to the desired values. A press on the “Reset to default Dates” button adjust all timelines to the default values.
- **Metric Selector** (C, left) allows a user to switch between individual metrics (one at a time). Upon selecting an option input, the metric algorithm runs on the fly and outputs a result table above the timelines (shown in Figure 5).
- **Issue Selector** (D, left) supports the filtering for issues in a project. The checkbox beside the issue key includes, respectively removes, a timeline for this issue.
- **Timeline Area** (E, middle) contains for each selected issue a separate timeline. A single timeline shows all events that occurred during the issues lifetime. By hovering over a single event additional information is displayed (see Figure 4).
- **Event Selector** (F, right) lists all the symbols, i.e., event types, that may occur on a timeline. Un/selecting a symbol hides/includes all corresponding event instances from all timelines.

5 QUALITATIVE EVALUATION

5.1 Study Design
We conducted semi-structured interviews with four stakeholders (i.e., actual end-users of the prototype): team-leads and group-leads.

Group-leads are responsible for the engineers in their group and assign work packages to them. Whereas team-leads are responsible for the group-leads and assign them projects and want frequent updates about project progress and workload. Each (separate) interview consisted of an introduction of the prototype including explanation of the metrics, used data, and user interface features. Subsequently, each participant was asked to assess three issues with respect to identifying coordination problems (Section 5.4). Thereafter, participants were asked to choose the three most insightful metrics (Section 5.5), score the user interfaces on a Likert scale from 1 to 5 stars (5 stars being best) along Jacob Nielsen’s heuristics [17] (Section 5.6), before providing free-form feedback on positive and negative impression as well as ideas for additional prototype features.

5.2 Data Set
The participants had access to data from four projects via the prototype within 30 minutes. These four real ACME projects are: P1, a low-priority Android app development project; P2 and P3, two business-critical Android app development projects; and project P4 integrating two types of recreational activities that involved experts beyond front-end, business logic, design, and database engineering (e.g., marketing and legal departments). The four projects P1–P4 contain a total of 1017, 2676, 1052, and 939 issues, respectively. The five most common issue types are Task, Bug, Improvement, Localization, and Project Management and make up between 80% and
5.3 Participant Demographics
The voluntary participant list consisted of two team-leads and two group-leads. The job title for both team-leads is “Teamlead WebDev”. Group-lead job titles were “Grouplead Architecture & Performance” and “Grouplead App” thus spanning a range of the industry partner’s engineering departments. The group-leads are in their position since 0.5 and 1 year, the team-leads since 1 and 10 years, having software engineering experience for 12 and 5 years, and for 6 and 17 years, respectively.

5.4 Task: Issue Assessment
Each participant received the same three issues selected from across three projects (P2, P3, P4; P1 is used for comparison only). They were asked to assess these issues using timelines and metrics (available

90% of all issues. During data extraction, the department prefixes in Table 1 were added to the anonymized engineer ids.
for the complete projects) but without no additional information from Jira (see Figure 6 for the timelines of the three issues: BETDB-1475, BAH-71, WWW-7370). The issues BETDB-1475 and BAH-71 were chosen, because they show problems in organization and workflow. The issue WWW-7370 represents a normal issue without anomalies. Since this task was part of an interview, we report the answers for the following two questions together:

- Can you spot any problem that occurred during the issues’ lifetimes?
- Can you identify high coordination efforts?

### BETDB-1475 (Figure 6 top)

BETDB is an issue primarily involving the database team. Every participant noted that status changes appear in quick succession from which they concluded (not only from this single issue but rather confirming their experience): the workflow for handling database issues is badly designed. When an SQL query needs to be fixed, nearly the whole workflow has to be executed again, except for an initial approval step. One problematic aspect, as mentioned by one of the participants, is that it is not clearly defined what should be done in the approval process. Furthermore approval does not seem to work if issues are re-opened that often. Participants remarked that the ill-design workflow is mitigated currently by the database team reacting very quickly on changes and immediately tackling them so that long delays are prevented.

Another hypothesis postulated by the participants is that requirements were not written well enough. This hypothesis is based on the fact that preparation (at the beginning of the lifetime) lasted one month and this is not the usual case. This is a problem because other teams have to wait. Also the issue was often re-assigned within the database team and it was also once re-assigned to a developer.

### BAH-71 (Figure 6 bottom)

For this issue all participants identified problems in how milestone and status fields are used and the overall lifetime of project issues. All participants stated that the lifetime is very long and it has to be asked if it was planned for that long at the beginning. Furthermore they mentioned that the issue was moved a lot to other milestones without any work in between and this indicates that issue planning is not efficient. Also status for issues are not handled correctly. Some are superfluous, e.g. “QA Test” which indicates that the whole project is in testing. However sub-issues are already tested before and the status does not make sense at project level. This problem is supported by the event on March 31st, where status changed from “In Progress” to “Internal Review” to “External Review” and then to “Softwaredesign”. A participant mentioned that it is not possible to review specifications of a whole project within a single day. Furthermore a project of this size should not stay one month in the design-stage, this is usually done faster. At the other end it seems that the whole project was tested in only three days which also cannot have occurred in reality. It was mentioned that the end of the timeline (beginning in May) represents the usual workflow.

### WWW-7370 (Figure 6 middle)

All participants declared that this is the usual workflow of a developer issue. This is the ideal case and it was probably a simple bug fix.

Participants identified the three causes for these specific issues mainly through timeline investigations. When the participants wanted to analyze the whole project they used the metrics. When a result of a metric was displayed they used timelines to analyze the outliers (e.g. issues with minimum or maximum metric values).

### 5.5 Top Rated Metrics

The next question asked the participants to select the three most valuable metrics for their work (see Table 2 GL = group-lead, TL = team-lead). We further report summarized participant statements regarding the usefulness and applicability of a metric together with the historical timeline visualization.

As shown in Table 2, every participant listed Metric 1 (Issue resolved/closed date <= issue due date) as one of the most valuable metrics. For the participants the metric is suitable for comparing projects and the timeline visualization can assist in answering the following questions:

- How long is an issue at a specific department?
- How was the project planned?

Metric 3 (Fix version wasn’t changed) was mentioned by 3 participants. It may indicate work overload and the visualization of the affected issue (and other contemporary issues) may provide insights to:

- Why were issues moved?
- How many issues are moved to the backlog or come from the backlog?
- Was an issue planned for an unrealistic time-frame?

Metric 5 (Re-open distance to due date), participants found, assists in finding out why issues were re-opened and the visualization supports detecting if there is a pattern for the re-open changes.

Metric 6 (Assignee Changes), participants noted, supports investigations if requirements were not formulated clearly or if perhaps task responsibilities were vague.

Metric 9 (Issue re-assignments without status changes), participants explained, highlights problems with long-running issues and visualization provides clues whether the workflow should be revised.

Metric 10 (Duration between re-assignment and subsequent status change), participants expressed, helps to identify poorly planned projects.

### 5.6 Usability Scores

The next part of the questionnaire asked the participants to rate the tool on a Likert scale of 1 (not good) to 5 (very good). Table 3 reports the chosen eight aspects (based on Nielsen’s heuristics [17]) and the average participant rating. Given the prototypical nature of our tool, the participants rated it very well across the eight issues.
Table 3: Average tool scores.

<table>
<thead>
<tr>
<th>Measure</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Responsiveness (performance with a large amounts</td>
<td>4.25</td>
</tr>
<tr>
<td>of data)</td>
<td></td>
</tr>
<tr>
<td>Forgiveness (allows exploration)</td>
<td>4.00</td>
</tr>
<tr>
<td>Intuitivity (easy to navigate)</td>
<td>4.00</td>
</tr>
<tr>
<td>Icons &amp; Symbols understandable</td>
<td>4.00</td>
</tr>
<tr>
<td>Information overload (sufficient/too much informa-</td>
<td>4.50</td>
</tr>
<tr>
<td>tion)</td>
<td></td>
</tr>
<tr>
<td>Low learning curve (easy to understand)</td>
<td>4.00</td>
</tr>
<tr>
<td>Match to the real world (terminology)</td>
<td>3.25</td>
</tr>
<tr>
<td>Flexibility/Efficiency of use (expert vs novice)</td>
<td>4.00</td>
</tr>
</tbody>
</table>

aspects, with exception to "Match to real world (terminology)". Here participants remarked that some metric names could be improved to more accurately convey their semantics.

5.7 Discussion

The answers to the two questions within the scope of the assessment task (see Subsection 5.4) revealed that metrics allow for quickly analysing a project and finding anomalies (participants had 10 minutes on average per issue). When investigating an issue in more detail, the participants always used the timeline visualization. The feedback on the tools usability (see Subsection 5.6) lets us assume that the Project Inspector prototype was sufficiently mature to assist the participants in their task. Most explicit negative criticism by participants concerned minor usability aspects such as tool tips, descriptions, font size.

In the free-form feedback part, participants explicitly highlighted the usefulness of combining metrics and timelines, quickly finding process flaws in short time, comparing projects, and the prototype’s simple design. Participants even suggested as future work to intensify this aspect by highlighting the events on the timeline that are relevant to the chosen metric.

Key Observation 1: switching back and forth between metrics and timeline is essential to quickly, easily obtaining insights into situations that would benefit from process improvement.

Participants also noted that for properly interpreting metrics and timelines, knowledge about how Jira is used by the teams is necessary: for understanding based on what data/events a metric is derived, but also to understand when engineers deviate from expected process behavior. During metric development our industry co-author but also the case study participants noticed that state transitions occurred often within unreasonable long or short time, implying that engineers were not accurately following the prescribed process. As potential future work, participants suggested to include additional coordination-centric meta information such as displaying the department of the engineer that made a status change.

Key Observation 2: an iterative cycle of metric selection, data collection, and metric interpretation is vital for ensuring that the metrics really measure something meaningful and for identifying additional useful metrics. Being able to flexibly integrate new or updated metrics is thus a key requirement, as enabled by our prototype.

The scoring of metrics (see Subsection 5.5) indicated that not all metrics were considered immediately useful during the assessment task. The most useful metrics either utilized changes to re-assignment of issue responsibility, state changes, or date changes. Participants mentioned in the free-form part of the questionnaire the lower-than-initially-expected usefulness of comment-centric metrics. They noted that the semantics of comments would need to be considered as well (most likely also other informal communication channels such as skype) and still potentially miss relevant
face-to-face communication during the occasional physical meetings.

**Key Observation 3:** Avoid, if possible, message content-based metrics (e.g., based on analysing comments, emails, etc.) that would require extensive coverage of multiple channels.

### 5.8 Threats to Validity

**Internal Validity:** We address researcher bias by analyzing data from an actual company rather than conducting controlled experiments. The approach works on arbitrary issue events and was not specifically tailored to Jira issues or the used dataset.

**External Validity:** Rather than claiming for wide generalizability of our results, we argue in line with Briand et al. [4] that context-driven research will yield more realistic results. In this paper, we thus evaluated the usefulness of the process metrics and issue timeline visualization with multiple engineers working at our industry partner. We analyzed data from this single company only as such data from real-world, industrial environments is extremely hard to get. Companies are very reluctant to provide insights into their working processes at that level of detail. As different engineers and roles (e.g., database expert, designer, team lead) have different concerns, they might evaluate the metrics’ usefulness differently. The positive feedback from multiple engineers familiar with the development processes under study, however, at least shows that the approach is applicable and indeed useful in the observed context.

We make no claim that our approach will yield equally useful results when applied to data from a Jira server used for open-source development. Often these servers, such as those hosted by the Apache Software Foundation [5] provide only the default issue types and issue states and thus require engineers to limit their coordination to processes based on this limited set of states, or apply other non-structured mechanisms such as comments, mailing-lists, and tacit knowledge to manage processes. Even less structured are issues in projects hosted on GitHub. GitHub issues [6] are either “Open” or “Closed”, any intermediary state needs managing via arbitrary labels (i.e., tagging) with no support for defining or restricting valid transitions. This restricts the ability to determine fine-granular metrics. The visualization prototype, however, is flexible to incorporate custom build metrics as long as they are provided with the JSON data items in the CouchDB.

Engineers assigned to departments and roles are a second, context-specific characteristic of the analyzed data set, Jira itself is unaware of roles (i.e., who should be changing an issue’s state) and hence such information cannot easily be extracted via its REST API. As many of the metrics derived from the data set include roles, we cannot infer how useful our approach will be for environments where no role information is available or where roles and departments are not clearly assignable. This will often be the case in open-source projects because they rarely exhibit a clear department and/or role assignment structure. Hence a comparison of the four projects, respectively the metric usefulness, to open source projects would make little sense.

### 6 RELATED WORK

Issue trackers have become an important tool for teams to coordinate their work. Managing the increased number of issues, however, has become a challenge [2] that multiple researchers aim to address.

Luijten et al. [15] introduced a tool to generate three different views that enable assessment of the issue handling process: a high-level (Issue Churn View), a quantitative (Issue Risk Profiles) and a detailed life-cycle (Issue Lifecycle View) view. Knab et al. [13] visualize the duration of a process step (submitted, in_analysis, in_resolution, in_evaluation) with a pie chart and provide a state transition view for problem reports.

Sarma et al. [20] proposed Tesseract, a socio-technical dependency browser that enables exploration of relationships between artifacts, developers, bugs, and communications, for example highlighting developers that are modifying interdependent code but are not communicating with each other.

Dal Sasse and Lanza [8] implemented in’Bug, a web-based software visual analytics platform. Extracting data from bug tracking systems, different panels describe high-level information such as duration (as a horizontal stacked bar chart) and status of bugs as well as fine grained views describing changes to a bug report’s properties.

Similar, D’Ambros et al. [9] focus on becoming aware of critical issues. Their “Bug Watch” visualization helps to understand the various phases that it traversed. They note that the criticality of a bug is not only dependent on its severity and priority but also on its life cycle. Frequently opened bugs indicate deeper problems.

Similar, D’Ambros et al. [9] focus on becoming aware of critical issues. Their “Bug Watch” visualization helps to understand the various phases that it traversed. They note that the criticality of a bug is not only dependent on its severity and priority but also on its life cycle. Frequently opened bugs indicate deeper problems.

Tužin et al. [23] describe their progress towards a unified process monitoring solution based on the Essence language and kernel. Also Brandt et al. [3] build on the Essence framework for project state visualization but focus on a Kanban style visualization rather than metrics and issue history.

Poncin et al. [18, 19] introduced the Framework for Analyzing Software Repositories (FRASR) for combining data from source code repositories, email lists, and bug trackers. They subsequently utilize the ProM process mining framework for obtaining insights such as classifying developers in open source software projects to roles such as project leader, core member, peripheral developer, bug fixer, or reader. They also analyzed typical bug report state transitions on Bugzilla.

Gupta et al. [11] conducted process mining across an issue-tracking system, a code review system, and a version control system. They map events from these systems into a single process (based on states) and determine transition occurrences. Based on this annotated transition diagram, they analyze the bug-fixing process from reporting to resolution to discover bottlenecks, deviations from the intended process, joint activities, and work handover.

None of these approaches provide a combination of (flexibly selectable) metrics and timeline visualization. In our previous work, we investigated an alternative approach to defining explicit metrics [14]. We applied constraints mining to issue histories from multiple projects to derive meaningful metrics for describing the software

---

1 https://issues.apache.org/jira
2 https://help.github.com/articles/about-issues/
development process. Such an approach is suitable to identify additional metrics for integration into our prototype.

7 CONCLUSIONS AND FUTURE WORK

In this paper, we presented an industrial case study and approach for supporting metric-driven process improvement. Specifically, we focused on coordination-centric metrics and consequently targeted process-related data and events in our evaluation prototype Process Inspector. Qualitative analysis with events and team-leads and groups-leads from our industry partner demonstrated that the combination of metric data with issue timeline visualization is a powerful approach to obtain process insights and quickly identify flaws in the process, inefficient coordination in issues, and comparing coordination aspects across projects.

As part of future work, we plan to improve the prototype along the received feedback, but more importantly, evaluate the use of the prototype across a project’s lifetime. This includes also evaluating to what extent these metrics are able to detect concrete differences when project teams are structured vertically as opposed to horizontally.

ACKNOWLEDGMENTS

This work was supported in part by the Austrian Science Fund (FWF): P29415-NBL funded by the Government of Upper Austria; and the FFG, Contract No. 854184. Pro²Future is funded within the Austrian COMET Program—Competence Centers for Excellent Technologies—under the auspices of the Austrian Federal Ministry of Transport, Innovation and Technology; the Austrian Federal Ministry for Digital and Economic Affairs and of the Provinces of Upper Austria and Styria. COMET is managed by the Austrian Research Promotion Agency FFG.

REFERENCES

5.5 Does the propagation of artifact changes across tasks reflect work dependencies?

Authors:
Christoph Mayr-Dorn, Alexander Egyed

Venue:

DOI:
https://doi.org/10.1145/3180155.3180185

Abstract:
Developers commonly define tasks to help coordinate software development efforts—whether they be feature implementation, refactoring, or bug fixes. Developers establish links between tasks to express implicit dependencies that need explicit handling—dependencies that often require the developers responsible for a given task to assess how changes in a linked task affect their own work and vice versa (i.e., change propagation). While seemingly useful, it is unknown if change propagation indeed coincides with task links.

No study has investigated to what extent change propagation actually occurs between task pairs and whether it is able to serve as a metric for characterizing the underlying task dependency. In this paper, we study the temporal relationship between developer reading and changing of source code in relationship to task links. We identify seven situations that explain the varying correlation of change propagation with linked task pairs and find six motifs describing when change propagation occurs between non-linked task pairs. Our paper demonstrates that task links are indeed useful for recommending which artifacts to monitor for changes, which developers to involve in a task, or which tasks to inspect.
Does the Propagation of Artifact Changes across Tasks reflect Work Dependencies?

Christoph Mayr-Dorn
Johannes Kepler University
Linz, Austria
christoph.mayr-dorn@jku.at

Alexander Egyed
Johannes Kepler University
Linz, Austria
alexander.egyed@jku.at

ABSTRACT
Developers commonly define tasks to help coordinate software development efforts—whether they be feature implementation, refactoring, or bug fixes. Developers establish links between tasks to express implicit dependencies that need explicit handling—dependencies that often require the developers responsible for a given task to assess how changes in a linked task affect their own work and vice versa (i.e., change propagation). While seemingly useful, it is unknown if change propagation indeed coincides with task links.

No study has investigated to what extent change propagation actually occurs between task pairs and whether it is able to serve as a metric for characterizing the underlying task dependency. In this paper, we study the temporal relationship between developer reading and changing of source code in relationship to task links. We identify seven situations that explain the varying correlation of change propagation with linked task pairs and find six motifs describing when change propagation occurs between non-linked task pairs. Our paper demonstrates that task links are indeed useful for recommending which artifacts to monitor for changes, which developers to involve in a task, or which tasks to inspect.

CCS CONCEPTS
• Software and its engineering → Software evolution;

KEYWORDS
task links, change propagation, bugzilla, mylyn, empirical study

ACM Reference Format:
https://doi.org/10.1145/3180155.3180185

1 INTRODUCTION
A task in software engineering defines a work item—usually for feature implementation, refactoring, or bug fixes. Often, tasks are broken down into subtasks that can be solved by individuals.

Throughout this paper we use the term task to represent any work item such as issue ticket, bug, change request, feature, or story.

In such cases, tasks can be a coordination mechanism to manage software development efforts. However, tasks may also arise out of other reasons. For example, dependencies among tasks commonly occur when closely related source artifacts are changed (e.g., methods that call one another or that share data) or when developers in one task wait for the output of another task. In such cases, the developers need to explicitly coordinate the involved tasks [17]. The developer responsible for a given task then has to not only understand the changes implied by the given task but also assess the impact of these changes onto dependent tasks. The developer responsible for a dependent task, in turn, studies these changes to assess the impact on his or her own work. Hence, we should expect that developers access artifacts in dependent tasks while working on their own—a practice we refer to as change propagation.

When creating tasks, developers face the challenge of identifying dependent tasks. One would assume that developers use the links offered by task management tools to make the implicit task dependencies explicit—links that may be incorrect and incomplete at times. They then use the links to identify which developers to notify/invoke about changes and which artifacts to change. Since identifying relevant engineers and artifacts for change propagation remains a significant problem [6, 7, 19], the question arises when do links actually reflect change propagation?

We believe insights into the usefulness and applicability of change propagation to identify task dependencies can provide an effective basis for novel support of developers during software change. Such insights are valuable beyond advising developers which linked tasks to monitor for changes [7]. They determine under which conditions change propagation metrics may detect implicit dependencies between non-linked tasks. Developers may then decide to link them, respectively monitor them for changes. Studying change propagation provides an understanding how a posteriori analysis of change propagation may result in reclassifying existing links. This reduces a developer’s effort to understand the implicit dependency between tasks and reveals further relevant tasks during future software evolution activities [6, 19]. A concrete scenario in Section 1.1 motivates the importance of the presented research and potential benefits for developers in further detail.

To obtain these insights we need to closely investigate to which extent and under which conditions change propagation correlates with links and whether non-linked task pairs exhibit similar behavior. To the best of our knowledge, no study has investigated change propagation across tasks. Existing work typically identifies coordination needs by determining which artifacts are usually changed together [11, 31, 32], how to correctly propagate changes among artifacts [2, 20], or how to detect inconsistencies [9, 10, 14]. Yet
non of the approaches consider the significance of links for change propagation.

This paper analyses the temporal relationship between developer reading and changing source code on the one hand, and task links on the other hand, as found in the Mylyn data set. Mylyn is an open source task management tool for the Eclipse IDE that captures traces of developer interactions (i.e., artifact reads and writes). Mylyn developers use this tool during their work on Mylyn. Mylyn, therefore, serves as the data gathering tool as well as the system under investigation in this paper.

Ultimately, we make the following four contributions:
(1) We identify seven situations that explain when linked task pairs exhibit change propagation;
(2) We identify six motifs that explain why non-linked task pairs exhibit change propagation;
(3) We lay out the implications of the found situations and motifs on future development support tools; and
(4) We provide a data set that combines developer interactions, tasks, and task links.

Specifically, we find that 64% of linked task pairs exhibit change propagation (true positives). The remaining 36% false negatives can be explained by three situations in which developers use links to manage tasks dependencies that do not entail change propagation. Examples are task synchronization and task decomposition. We identify additional four situations that describe distinct artifact-centric task dependencies. Artifact reuse or work continuation dependencies, for example, explain why task pairs exhibit strong change propagation. Further analysis showed no change propagation for 93% of all non-linked task pairs (true negatives). We discover six motifs that enable the classification of the remaining 7% false positives as either true positives (i.e., task pairs that should have been linked) or true negatives (i.e., task pairs with irrelevant change propagation). These insights are vital for designing recommendation mechanisms that utilize change propagation, for example, to trigger change notifications between tasks, respectively, identify relevant tasks.

The remainder of this paper is structured as follows: We refine our general research hypothesis and present our study design in Section 2. We analyse the Mylyn data set quantitatively in Section 3, Section 4 and 5 detail the manual inspection of sample linked and non-linked task pairs, respectively. We interpret our findings and implications in Section 6. We discuss related work in Section 7 before concluding this paper with an outlook on future work in Section 8.

1.1 Motivating Example
We motivate the need to investigate change propagation between task pairs using an actual example task subset from the open source Mylyn project. Mylyn [15] allows a developer to connect to a task management tool (such as Bugzilla) for selecting tasks to work on and captures all developer read and write events within the Eclipse IDE. The tasks in our example address different mechanisms for creating a new Mylyn task. Figure 1 depicts the links among tasks as of Nov. 14, 2007. The central Task 169426 has links to five tasks. The greyed out Task 210022 has not been set up yet. Task 209892 (bold) was just created and thus no progress has been made yet. All tasks are in status "open". As the developer S.P. assigned to Task 209892 commences work, he needs to know where to look for artifacts and their (recent) changes relevant to the realization of his task. Likewise, the developers currently working on the other open tasks need to assess who they should work with and perhaps notify about changes.

Without a support tool, developers need to maintain an up-to-date view on what is going on in each linked task, a very tedious, time-consuming, and error-prone process as small details are easily missed and links may be inaccurate or incomplete. In the month prior to Nov. 14, 2007, there are 59 developers accessing ~1500 artifacts in 164 tasks. Alternatively, developers may choose to observe only the directly linked task pairs and miss important developments in other tasks. Access to change propagation information—which as displayed in Figure 1—may serve as indicator what tasks are relevant. S.P. may deduce from the change propagation values that Task 161646 shares artifacts not only with the central task, but also with Task 209402 and subsequently monitors primarily these two for changes. Yet, it is currently unclear to what extent links between tasks describe implicit task dependencies relevant to change propagation and, hence, whether one could reliably exploit them to determine where changes should be propagated to, respectively to notify about which particular change. As we mentioned above, no study has investigated the correlation of change propagation and task links.

Understanding how change propagation occurs between linked task pairs is also important for supporting software evolution activities. Suppose we now encounter a task for fixing a bug related to incomplete cloning of data in the Mylyn task editor. The responsible developer may identify Task 209402 perhaps through keyword search, referral in the task’s comments by another developer, or vaguely remembering that it once concerned data cloning. No matter how, she then needs to understand if that Task 209402 covers the problem, or if the linked Task 169426 is relevant also, or if any of the other indirectly linked tasks need inspection. Applying past change propagation to better classify the links among the tasks assists the developer to quickly narrow down the relevant task locations. Again, this requires an informed understanding how links among tasks coincide with change propagation.

Figure 1: Example excerpt of linked task pairs. Full lines depict manually set links. The dotted line displays change propagation among non-linked task pairs. Line labels report the number of propagated, changed artifacts.

398
2 STUDY DESIGN

2.1 Research Questions
This paper aims to answer the following research questions:

**RQ1**: What is the extent of change propagation that occurs between linked task pairs? This question focuses on whether artifacts that are subject to a change in one task are eventually accessed in the linked task and vice versa. We analyze what proportion of changed artifacts are accessed and how much linked task pairs differ from non-linked task pairs.

Answering this question gives us insights into whether linked task pairs represent a change propagation dependency, respectively. Whether change propagation indicates a need for coordination (i.e., creating a link between two tasks) or, if there is no (consistent) change propagation among linked task pairs, then developers reliably use links for deciding who to involve in a task or notify about changes.

**RQ2**: What are the reasons for change propagation that occur between two tasks? This question extends the previous question and investigates under which conditions are linked task pairs showing change propagation (true positives) and when there is no change propagation occurring (false negatives). Additionally, this question studies the cause for non-linked task pairs to exhibit change propagation (false positives).

By identifying the cause for change propagation, we are able to determine more precisely when propagating changed artifacts among linked task pairs is relevant and whether—highly important for providing effective change notifications. Identifying the reasons also allows us to identify dependent but non-linked task pairs—this is potentially indicating a missing link.

2.2 Data Gathering Method
The Mylyn project uses the Eclipse Bugzilla bug tracker for managing task dependencies. Developers working on Mylyn attach the captured read and write events (i.e., the interaction data) to the tasks they are responsible for. Hence, for this project, we know what task a given developer was working on and what artifacts he or she was looking at or modified. In this paper, we are interested in changes at the file level.

We extracted 410 tasks with attached interaction data and at least one Bugzilla blocks/depends_on link - referred to as the base set. Those 226 tasks that have a link to another task in the base set form the linked set. We end up with a total of 160 links in the linked set.

The supporting online material [18] provides (i) a more detailed description of the data gathering process, (ii) the set of tasks and attachments considered, (iii) the source code for collecting, filtering, and analyzing the data, as well as (iv) the aggregated data underlying all figures and tables in this paper.

2.3 Temporal Data Processing
The temporal order of interaction events is important to accurately determine which changes in one task could have been accessed later—potentially by the same developer—in the linked task. Figure 2 visualizes this procedure.

We group interaction events into interaction sessions as developers tend to commit changes when they completed part of a task rather than instantaneously. Figure 2 depicts example task a’s three sessions (s1, s2, s3) on the top and task b’s three sessions on the bottom. Session duration is represented by horizontal size, with the read-only (r) and written (w) artifacts identified by number, e.g., in session s1 a developer changes artifacts 1 and 7 and reads artifact 2. By default, each interaction data attachment becomes one session. We split long lasting sessions whenever two events are more than 1.5 days apart.

We make the assumption that a change (as recorded by a write event) in one interaction session potentially propagates to all subsequent sessions in the linked task. The dashed lines in Figure 2 highlight the potential change propagation direction. The PCS set contains all artifacts that a developer changes in one session of task a and which subsequently a developer accesses (i.e., read or write event) in any subsequent session of task b. For example, a change to artifact 1 in session s1 is read by a developer in session s2 who subsequently changes the artifact in s3. There is no propagation across parallel or partially overlapping sessions. Changes to artifact 2 in session s2, for example, don’t propagate to task b as session b3 happens at the same time. The complete set of artifacts changed in task a and propagated to task b (CPa→b) is the union of all \( P_{i=1}^n \) for each pair of artifacts (CPa→b). This guarantees that we count a propagated artifact only once, even when it is accessed in multiple sessions. Table 1 summarizes our formalization of change propagation.

The absolute number of propagated artifacts (CPa→b) tends to overestimate the importance of change propagation between tasks where the developer accesses a large number of artifacts (e.g., task b in Figure 2). The more artifacts are changed within a task, the more likely these are accessed later. Conversely, the more artifacts a developer accesses, the more likely these artifacts were previously changed in the linked task. The resulting high number of propagated changes as measured by absolute change propagation (CPa→b), however, might be coincidental to the dependency among the two tasks. On the other hand, a developer working on a small task might be interested only in a subset of all changed artifacts (e.g.,

---

3https://www.eclipse.org/mylyn/
4https://bugs.eclipse.org/bugs/query.cgi
The Bugzilla duplicates link type is irrelevant for this paper.

---

5We opted for this large window as open source software developers often work in their free time and may spread work across several days to complete a task.
task a in Figure 2). This results in low absolute change propagation and risks underestimating the relevance of the propagated changes.

We, therefore, introduce two metrics that mitigate the limitations of absolute change propagation. Specifically, we introduce the Relative Observed Change Propagation (ROCP(a, b)) defined as the ratio of propagated artifacts (CP_{a,b}) to all changed artifacts in task a (ROCP(a, b) = |CP_{a,b}|/|WA_a|). A value of 1 indicates that the developer responsible for task b accessed all artifact changed in task a, whereas a value of 0 implies that the developer accessed none of the changed artifacts. Thus, Relative Observed Change Propagation ensures we consider the propagated changes only as relevant when the developers of task b accesses close to all changed artifacts, but not necessarily when accessing a lot of previously changed artifacts.

The Relative Attainable Change Propagation metric (RACP(a, b)) is defined as the ratio of propagated artifacts (CP_{a,b}) to all accessed artifacts in task b (RACP(a, b) = |CP_{a,b}|/|WA_b|). A value of 1 signifies that all the artifacts a developer accesses in task b have been changed in task a before, whereas a value of 0 implies that the developer accessed only artifacts that have not been changed in task a before. Thus, Relative Attainable Change Propagation ensures we consider even a small set of propagated changes as relevant when a developer accesses a few artifacts, most of which have been changed in the linked task.

We combine ROCP(a, b) and RACP(a, b) to obtain the Relative Change Propagation (RCP(a, b)) from task a to task b: RCP(a, b) = (ROCP(a, b) + RACP(a, b))/2. Ultimately, the sum of RCP(a, b) and RCP(b, a) produces the Bi-directional Relative Change Propagation (BiRCP(a,b)) between tasks a and b. A value of 0 indicates that absolutely no change propagation occurred in either direction.

value of 2 indicates that developers accessed and changed exactly the same set of artifacts in two concurrent tasks (a very rare case).

Examples in Section 4 show that BiRCP is better suited for determining whether two non-linked tasks are dependent than bi-directional absolute change propagation (BiCP).

### 3 QUANTITATIVE ANALYSIS

In this section, we quantitatively analyse the Mylyn data set to provide an answer to RQ1. We calculate BiRCP for all linked task pairs in the linked set. The histogram of these BiRCP values (see Figure 3 (light grey), Table 2 left) shows that 58 linked task pairs (~36%) don't exhibit any change propagation. The other 102 task pairs cover the change propagation spectrum up to 1. On average linked task pairs show 21% BiRCP (σ = 22%, median=16%) and 1.81 BiCP (σ = 2.36, median=1).

We also inspect the change propagation among non-linked task pairs from the same data set in order to confirm that the extent of change propagation can be attributed to the links among tasks and not to a general feature of the data set. To this end, we calculate the same change propagation metrics (see Table 1) for all task pairs in the base set that are more than two link-hops apart - denoted the non-linked set. I.e., we exclude task pairs that have links to a common third task as we expect several of these pairs to yield similarly high change propagation as directly linked task pairs. We report the resulting BiRCP values in Table 2 (right) and Figure 3 (dark grey). Around 93% of the 82,129 pairs in the non-linked set exhibit no change propagation. On average non-linked task pairs exhibit 1.14% BiRCP (σ = 5.35%, median=0%) and 0.12 BiCP (σ = 0.66, median=0).

Comparing BiRCP average, standard deviation, median, and the histogram distribution in Figure 3, we conclude that change propagation is a feature of the linked set. We, however, find two orders of magnitude more task pairs in the non-linked set with non-zero change propagation (5897) compared to the linked set (102 pairs with BiRCP > 0).

We draw the following preliminary conclusions: First, based on the quantitative analysis alone, linked task pairs imply an underlying change propagation dependency (~64% true positives). Hence, a recommendation algorithm may utilize the links between two tasks as an indicator that developers in one task, for example, should be notified about changes in the linked task. Yet, from the distribution of BiRCP values we learn that there is a significant number of task pairs where no change propagation occurs (~36% false negatives) and thus recommendations would be irrelevant. We subsequently use the underlying quantitative data to sample task pairs for qualitatively investigating the reasons for high, respectively, low BiRCP in Section 4.

Second, the large number of task pairs with non-zero change propagation in the non-linked set (~7% false positives) entail that randomly selecting two tasks from the base set and detecting change propagation provides no indication whether the two tasks might indeed be dependent. In other words, the data imply that a recommendation algorithm suggesting related tasks based on change propagation alone will very likely produce a list of tasks that are not truly relevant. Reducing the number of false positives is pertinent.

We, thus, use the underlying quantitative data to sample task pairs
for qualitatively investigating the reasons why some non-linked task pairs exhibit high change propagation (see Section 5).

4 QUALITATIVE ANALYSIS OF LINKED TASK PAIRS

In this section we aim to answer RQ2 with respect to determining the reasons for high and low change propagation among linked task pairs using methods from Grounded Theory [27]. First, we sampled 17 task pairs from the 160 links in the linked set (see Table 3). We applied following sampling criteria: select a combination of zero and non-zero BiRCP with small as well as large write sets ($W_i$). For each task pair, we manually inspected the task details, task description changes, and comments on the Eclipse Bugzilla website. We captured all information pertinent to work coordination among tasks on virtual cards in an open coding process [27]. Upon completing the processing on all samples, we iterated through the cards and retained those that represented common coordination concerns. Based on these remaining cards, we aimed to establish the purpose of the link between two tasks, and thus the reason for low or high change propagation.

The Tasks 169426 (Sample 3 and 6) and 200634 (Sample 10) are part of a larger graph of linked tasks for which we have interaction data. In a second round, we included all tasks in these two graphs in our analysis (increasing the manually analysed task pairs to a total of 25, i.e., ~16% of all links in the linked set). In total, we inspected 41 out of the 226 tasks in the linked set. Page restrictions limit us to a brief introduction of the two connected task graphs. We subsequently map the 25 task pairs to the identified coordination concerns (see also right most column in Table 3). When referring to a sample task pair, we indicate the task’s location in Table 3 with

4https://bugs.eclipse.org/bugs/show_bug.cgi?id=TASKID

401

Figure 3: Histogram of the relative amount of task pairs from the linked set (light grey) and the non-linked set (dark grey) as determined by BiRCP bins. Note that the percentage on the log-scale $y$-axis is given to 3 decimal places.

Figure 4: Artifact change propagation graphs for two selected cases: tasks are represented by boxes that report the number of changed artifacts, their size indicates work duration, time flowing from left to right. The dark shaded box identifies the central task that all other tasks link to. Lines report BiRCP\(|a||b|\).


4.1 Connected Sample Graphs

We introduced TaskGraph 1 in the motivating scenario (Sec. 1.1). The upper three tasks represent desirable features that have been sitting dormant for several months. Work on Task 161646 becomes the basis for the development in the central Task 169426. Figure 4 (top) visualizes how the work in the remaining tasks occurs in parallel to the central task. During development, subtasks 209402 and 209492 emerge that aim to provide a reusable interface for the other tasks to program against. Task 207524 represents another related feature, limited to a subcomponent (i.e., the Bugzilla client). Task 210022 continues the work of Task 152869 by fixing a bug introduced there.

TaskGraph 2 addresses the ability to obtain authentication details from the user. Similar to TaskGraph 1, significant up-front work is done in Task 200634 which serves as basis for refinement and extension (see Fig. 4 bottom). In contrast to TaskGraph 1, however, here the up-front work task becomes the central task, and all remaining linked tasks represent various subtasks. The first two subtasks (207521 and 207527) concentrate on the two distinct efforts to avoid storing a password and the ability to retrieve the password on demand from the user, respectively. The next two subtasks (207531 and 207654) integrate the capabilities of the prior
two task in two different connectors to third party systems (Trac and JIRA). Work in the last task (210483) occurs several weeks after the previous tasks have been completed and undertakes rework of existing artifacts to improve usability.

4.2 Situations leading to low BiRCP

We found three main situations that lead to low Bi-directional Relative Change Propagation.

S1: Task Decomposition One task serves as parent task for multiple child tasks which refine the work described in the parent task. When the parent task is predominantly used for coordinating work and subsequently involves little coding effort and thus few changes, then there is little opportunity for change propagation.

For example, Task 228822 (b) in Sample 2 serves as central parent task coordinating the refactoring of attachment data handling. The linked Task 228833 is one among a few child tasks realizing the refactoring. The task pair exhibits low BiRCP as the parent task includes only a few, intermittent changes. See, e.g., one comment in Task 228822:

Remaining work:
- set defaults for the task editor or working copy
- replace TaskSelection with new API

S2: Separation of concerns applies to task pairs that conceptually belong together (hence the task links) but address different aspects of the same concept and thus—regardless of the number of changed artifacts—share few changed artifacts as work in the respective tasks tends to affect different artifact sets.

We find such situations in TaskGraph 1—Task 207524, 209402 (a) (i.e., Sample 6), and 209892 all linked to Task 169426 (b)—and in TaskGraph 2; here in Sample 10 Task 207531 (a) links to Task 200634 (b). In Sample 4 both tasks are related refactorings postponed to a different release. The Tasks 191793 (a) focuses on code refactoring, the linked Task 202547 (b) focuses on feature restructuring of non-code artifacts.

S3: Synchronization aims at coordinating work among task pairs that are likely to change the same artifact and cause merge conflicts. Typically only a few central artifacts are subject to a potential write conflict and thus BiRCP remains low.

We find such a case in Sample 5: Work in both tasks involves changes to a central file (the only artifact changed by both tasks). The link serves as locking mechanism to avoid a conflict i.e., first completing Task 244653 (a) and subsequently the linked Task 242445 (b). Note following excerpt from the comments of Task 242445:

Helen, it is probably best to wait until the patch on bug 244653 is merged before starting on this to avoid conflicts.

4.3 Situations leading to high BiRCP

During our analysis we identified four situations that give rise to high BiRCP.

S4: Work Continuation occurs when developers in one task explicitly hand over development effort to another task. Work in the continuing task is then likely to change the same artifacts and hence lead to high BiRCP.

In Sample 9 we find developpers in Task 196700 (b) implementing a new feature. This feature contained a bug that was subsequently corrected in the linked Task 201464 (a). The former task thus continued and completed the work of the latter task, hence the large BiRCP. See following comment in Task 196700:

Warning! There appears to be some corruption [...] there is another bug to track this issue: bug 201464. I would avoid using the new task editor feature on any kind of production scheduler server until that bug is fixed.

In TaskGraph 1, work on Task 169426 (b) and 209892 continues on code from Task 161646 (a); in the comments for Task 169426 (b) we find:
Does the Propagation of Artifact Changes across Tasks reflect Work Dependencies? ICSE ’18, May 27–June 3, 2018, Gothenburg, Sweden

Thanks for the patch Frank. It would be great if you could make the implementation more generic so it could be reused for bug 161646 which overlaps significantly with this bug.

In TaskGraph 2, Task 210483 continues the work of streamlining the artifacts of preceding tasks 207521, 207527, 207531, and 207654.

S5: Artifact Reuse resembles S4: Work Continuation but differs in the explicit focus on making use of the output of a linked task, rather than continuing the work.

In TaskGraph 1, work on Task 161646 is explicitly postponed to continue after the output of linked Task 209402 is available: We’re holding off on this until we can make use of the api that emerges from bugs/268482.

In TaskGraph 2, developers in Tasks 207521 and 207527 reuse code of the parent Task 200634 as the comment in Task 200634 highlights.

[...]

In Sample 16, Task 231336 (a) fixes a sorting problem and provides code to be used in the linked Task 216150 (b) that is equally concerned with sorting. A comment excerpt in Task 231336 states: [...] for bug216150 the compare part is now in class TaskComparator [...]

S6: Emerging Task Decomposition results in several tasks becoming the children of the common, central linked task (similar to S1). However, here, significant up-front work is done in the central task and the child tasks are created one by one as needed.

In Sample 7, a significant amount of work occurred in Task 238038 (b) (about providing time range based folding for comments) when the linked Task 244359 (a) was specified as a subtask to implement a sub-aspect of the parent (implement grouping strategy for task comments). Hence, the BiRCP among linked task pairs is high. The central task exhibits further links to other subtasks that were created step by step as the work progressed. In the emerging child Task 244359 we find following comment:

Extract the implementation for grouping of task comments discussed on bug 238038.

In Sample 8 the two tasks address support for test integration with third party systems. In Task 386344 (b), the work is done for one third party system (here Trac). In the linked Task 393640 (a), the work is then replicated for another third party system which explains the high BiRCP. In Sample 13, Task 217694 (a) addresses which icon to use inside a tooltip while the linked Task 205861 (b) coordinates several tasks on improving tooltip presentation. Similar, in Sample 15, Task 303431 (a) is about colors, the linked Task 199345 (b) ties together tasks about configuring labels. Finally, Sample 17 has Task 277191 (a) focusing on UI issues in the connector discovery dialog and the linked Task 232621 (b) coordinating the Mylyn connector discovery mechanism.

Emerging Task Decomposition is closely related to S5: Artifact Reuse. If we restructure TaskGraph 1 to set Task 161646 as the parent task, and have Tasks 169426, 209402, and 209892 as child tasks, then these three child tasks would constitute examples of emerging task composition rather than simply artifact reuse.

S7: Small Bug Fixes involve corrections to a handful of artifacts that are changed in both tasks resulting in high BiRCP. In Sample 11, both tasks are related bugs about hyperlink processing which were independently rectifiable. Task 176212 (b) constituted minor corrections to only two artifacts which were subsequently changed again during work on the linked Task 229014 (a). Similar, the tasks in Sample 12 are two related bugs, here, one about the size, the other about the position of a user interface element. The single changed artifact in the Task 261683 (b) was changed again in the linked Task 262107 (a). Also Sample 1 fits this situation. Task 208629 (a) fixes a small bug, hence only a few changes, which largely overlap with the refactoring done in the linked Task 179524 (b). Sample 9 could also be classified as S7: Small Bug Fix, but comments and the link to a feature development task better places it with S4: Work Continuation. Finally, in Sample 14, Task 210170 (a) fixes tooltip visibility, relevant for improving tooltip positioning in the scope of the linked Task 189313 (b).

5 QUALITATIVE ANALYSIS OF NON-LINKED TASK PAIRS

In this section we complete our answer to RQ2. In Section 3, we found that a significant amount of non-linked task pairs exhibit high BiCP and/or high BiRCP. Here we manually inspect sample non-linked task pairs to determine whether high BiRCP is indicative of missing links or whether we simply detect a lot of false positives. To this end, we sampled 22 non-linked task pairs from the 5897 links in the non-linked set that have non-zero BiRCP, a total of 32 tasks. We applied following sampling criteria: select a combination of small as well as large write sets (Wj), a range of 1 to highest observed BiCP, and a range of 0.1 to 1.5 BiRCP. We followed the same coding procedure of the qualitative analysis of linked task pairs (see Section 4) with a focus on the set of propagated artifacts (CPa,b), any available links to other tasks, and comments. We cannot introduce the tasks individually due to page limits.

We identified six motifs that describe when task pairs exhibit high BiCP and/or high BiRCP and which explain when the these metrics uncover a true (albeit implicit) dependency among the tasks.

M1: Task Cluster Membership We find several cases where the tasks address the same implementation topic, e.g., hyperlink issues in Sample 12 and 114 or connector discovery in samples 14 and 115. Typically these tasks have links to other tasks that are directly linked. Recall that we excluded two-hop related task pairs in the quantitative analysis. Figures 5 and 6 (left) depict how these samples are three hops (I4, 115) or more (I2, 114) apart but still part of a cluster of related tasks.

M2: Support Cluster Membership Similar to M1, we find several tasks that focus on implementing and/or testing a particular feature or subsystem. Figure 6 (right) visualizes how change propagation among the tasks in Samples 19, 111, 116, and 118 ties together tasks on supporting a new version of Bugzilla that otherwise are not explicitly linked. Similar, Sample 16 brings together two dependent tasks: one fixing a new feature, the other provisioning the respective testing infrastructure.

M3: Mutual Access of Utility Artifacts Task pairs exhibit high BiRCP when they primarily read or update commonly used utility artifacts. All tasks in Samples 18 and 117 require UI features to be configurable. While the tasks focus on different UI elements (here labels, line highlighting, and search characters), they all needed to make changes to the same set of UI and preference-centric artifacts.
| Nr | Task_a | Task_b | |Ra| |Wa| |Rb| |Wb| |CPa,b| |CPb| |BiPC| |BiRCP| |M. |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|11 | 208629 | 220688 | 1 | 2 | 0 | 1 | 1 | 1 | 2 | 1.42 | M5 |
|12 | 164231 | 176212 | 2 | 4 | 1 | 2 | 2 | 4 | 1.25 | M1 |
|13 | 206568 | 216477 | 7 | 10 | 0 | 1 | 1 | 1 | 2 | 1.08 | M5 |
|14 | 274942 | 277910 | 3 | 5 | 1 | 1 | 1 | 1 | 2 | 1.00 | M1 |
|15 | 167741 | 208629 | 16 | 19 | 1 | 2 | 3 | 1 | 4 | 0.84 | M5 |
|16 | 201465 | 195623 | 6 | 4 | 2 | 2 | 3 | 0 | 3 | 0.75 | M2 |
|17 | 219911 | 175922 | 2 | 4 | 5 | 2 | 17 | 3 | 3 | 6 | 0.73 | M4 |
|18 | 199345 | 299697 | 71 | 67 | 6 | 6 | 5 | 5 | 10 | 0.68 | M3 |
|19 | 252297 | 256505 | 5 | 9 | 4 | 3 | 5 | 0 | 5 | 0.63 | M2 |
|20 | 206034 | 195656 | 51 | 28 | 7 | 16 | 1 | 11 | 12 | 0.45 | M4 |
|21 | 226851 | 242840 | 34 | 28 | 51 | 52 | 14 | 3 | 17 | 0.37 | M2 |
|22 | 149311 | 160399 | 52 | 57 | 150 | 101 | 1 | 19 | 20 | 0.19 | M6 |
|23 | 220688 | 216477 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1.00 | M5 |
|24 | 244442 | 164221 | 10 | 3 | 2 | 4 | 2 | 2 | 4 | 0.83 | M1 |
|25 | 276942 | 278931 | 3 | 3 | 1 | 3 | 1 | 2 | 3 | 0.79 | M1 |
|26 | 242480 | 255695 | 55 | 57 | 0 | 2 | 1 | 2 | 3 | 0.77 | M2 |
|27 | 199345 | 318045 | 71 | 67 | 6 | 4 | 6 | 2 | 8 | 0.53 | M3 |
|28 | 152972 | 244840 | 5 | 9 | 51 | 52 | 8 | 1 | 9 | 0.53 | M2 |
|29 | 178207 | 290265 | 45 | 61 | 17 | 30 | 4 | 13 | 17 | 0.35 | M6 |
|30 | 200634 | 160399 | 51 | 28 | 150 | 101 | 0 | 15 | 15 | 0.17 | M6 |
|31 | 238038 | 236490 | 59 | 12 | 124 | 55 | 0 | 14 | 14 | 0.23 | M6 |
|32 | 226780 | 175922 | 185 | 27 | 52 | 17 | 0 | 12 | 12 | 0.38 | M6 |

Table 4: Sample non-linked task pairs selected for qualitative analysis. The M. column lists the respective motifs M1 to M6; for metric definitions, see Table 1.

6 DISCUSSION

Manual inspection of linked task pairs revealed seven recoucuring situations. These explain why Bi-directional Relative Change Propagation coincides only with a subset of linked task pairs. Specifically, we find medium to high BiRCP (i.e., >0.2) only when links represent an underlying artifact-centric task dependency. We infer that developers use links to manage control flow, data flow, task decomposition, and simultaneity dependencies [17]. These situations are highly relevant for the design of development coordination support tools and thus have significant potential impact on software engineering practice. Qualitative analysis shows evidence that high BiRCP is able to identify implicit dependencies among task pairs when additional information on artifacts (e.g., core or utility) and tasks (e.g., refactoring) is considered. Absolute bi-directional change propagation is less well suited as it identifies mostly tasks involving many artifact changes.

M4: Orthogonal Concerns Tasks tend to access the same artifacts when these encode overlapping topics. Task a in Sample 17 focuses on setting certain properties when moving a bug to another product; task b focuses on supporting custom fields in Bugzilla. The propagated artifacts implement Bugzilla data handling functionality. Similar, both tasks in Sample 110 are related to connecting to Bugzilla for synchronizing data: task a addresses password acquisition, task b addresses data fetching.

M5: Core Artifact Access In several samples, one or both tasks access only a very small set of one to three common artifacts. This results in high BiRCP but no real underlying task dependency when these artifacts are at the core of the system under development and thus subject to many unrelated changes. All tasks in Samples 11, 13, 15, and 113 propagate changes of the same core artifact (AbstractRepositoryTaskEditor). The tasks, however, are otherwise completely unrelated.

M6: Grand Tasks We inspected task pairs with high BiCP to check whether abstract instead of relative change propagation is a suitable indicator for dependent tasks. Only tasks accessing many artifacts are able to yield high BiCP and subsequently are less likely to also yield high BiRCP as Table 4 shows. Task pairs with BiCP > 11 tend to exhibit comparatively lower BiRCP (< 0.45) and vice versa. All tasks in Samples 112, 119, 120, 121, and 112 access a high number of artifacts (|Ra + Wb|) ranging from ~20 to ~250. In these samples, one task is typically a refactoring effort while the other task is an unrelated, large feature implementation or bug fix.

Figure 5: Examples of Task Cluster Membership for sample non-linked task pairs. Dotted lines report BiPC and BiRCP, full lines depict links among tasks, dashed lines depict task references in the comments.
task. Rather, they should receive a warning when they are about to make any changes as developers of the follow-up task are not expecting changes. Links in situations $S_5$, $S_6$, and $S_7$ indicate that developers in either linked task are interested in an overlapping set of artifacts and hence would benefit from change-propagation centric coordination support.

The motifs found among the non-linked task pairs with high $BiRCP$ suggest that developers would benefit from coordination support beyond directly linked task pairs. M1. Task Cluster Membership demonstrates that change propagation and multi-hop paths across existing links identify dependent task pairs that, for example, serve as input to a change notification mechanism. Even in the absence of multi-hop paths, change propagation identifies dependent tasks in M2: Support Cluster Membership and M3: Mutual Access of Utility Artifact. In the case of M3 identifying non-linked task pairs assists the developer in finding examples how to use a particular artifact, suggestions on what artifacts to change as well, and recovering design justifications from comments.

The motifs highlight that change propagation alone is insufficient to reliably determine dependent (but non-linked) task pairs. M5: Core Artifact Access points out the need to identify frequently changed artifacts (i.e., the core artifacts) and assign less importance to them when measuring change propagation. Doing so reduces the likelihood of detecting false positive task pairs. Similar, M6: Grand Tasks suggests to ignore non-linked task pairs that exhibit a high amount of accessed artifacts and high absolute $BiPC$. The non-linked samples classified as M6 motivate a $BiRCP$ threshold of 0.4. This reduces the non-linked sample pairs under consideration for implicit dependencies from ~5800 down to 436. Note that this threshold is higher than the $BiRCP$ values we find in situations with high change propagation among linked task pairs. Some task pairs in situations $S_4$ and $S_6$ exhibit $BiRCP$ between 0.2 and 0.3. As we mentioned, the change propagation metrics alone are often not sufficient to reliably interpret a link. In the case of Samples 3 and 7, the manually inspected context determines their classification. Currently, the number of manually investigated samples is too low to infer the distribution of situations and motifs across the complete Mylyn data set and thus to reliably select a threshold.

Change propagation metrics are also useful for a-posteriori (re-) classification of links. Accurate and complete links are important during maintenance efforts as the basis for identifying relevant tasks to inspect. Correcting a bug requires identifying tasks that fixed a similar bug or similar location before (i.e., $S_7$). Developers engaging in refactoring may benefit from knowing where work has continued from an initial task and studying the latter for extracting design rationale ($S_4$ and $S_5$). High change propagation among child tasks in situations $S_1$ and $S_6$ assist in identifying closely related tasks, separating them from tasks that implement independent concerns ($S_2$).

6.1 Threats to validity

6.1.1 Internal Validity. We address researcher bias by analyzing data from an open source system rather than conducting controlled experiments. The analysis focused on artifacts and tasks and was not specifically tailored to Java development in general or the Mylyn project in particular. The manual inspection during qualitative analysis showed no indication that the use of links was specifically adapted to the Mylyn development "process".

With respect to the data set quality, we noticed that occasionally interaction attachments appeared to be missing (e.g., a long interval between interaction data attachment and commit message in the comments). The impact on our results, however, is minimal as missing attachments occurred typically towards the end of a task that contained several other preceding attachments. We thus expect the missing attachments to contain little additional information.

6.1.2 External Validity. We analyzed only a single data set as we are not aware of other real world projects aside from Mylyn that make a significant amount of interaction data and linked tasks available. Mylyn interaction data upload capabilities are not available by default and thus not widely used beyond the Mylyn project. Hence, we are careful to generalize our findings beyond the scope of the Mylyn project.

We can infer from other metrics; however, that the Mylyn project is similar to other projects. Thompson et al. [28] analyze the task links of three open source projects including Mylyn and finds the
ratio of links classified as Specification or Problem similar. Zou and Godfrey [33] report on maintenance tasks and finds a median of 2 for edited files and the median of viewed-only files in a task is 4. We found a median of 3 and 6, respectively, for Mylyn. D'Ambros et al. [5] report similar commit transactions per class and change coupling metrics for Mylyn and ArgoUML. Heck and Zaidman [12] show that Mylyn has similar duplicate bug reports as other open source projects. All indications are that Mylyn data are in fact transferable. We don’t expect all situations and motifs to arise in other projects, nor do we claim completeness, but the identified dependencies are found beyond the software engineering domain [17].

The Mylyn dataset might not accurately represent (non-open source) development environments where other communication channels (such as direct messaging or face-to-face discussion) exist for conveying the impact of an artifact change [6]. This reduces a developer’s need to inspect (i.e., read only) artifacts. We speculate that fewer observed read-only events subsequently result in lower levels of change propagation. Additional data sources and analysis are necessary to assess the effect of read-only events on change propagation in these environments.

Bugzilla limits link types to duplicates and blocks/ depends_on while other task management tools (e.g., JIRA, Redmine) provide diverse and customizable link types. The available link types, however, have no impact on the results as our analysis is independent of the link type semantics and direction. The identified seven situations reflect generally applicable coordination concerns that are common in software development and in no way specific to Bugzilla or the Mylyn project. We suspect that data from a project using JIRA or Redmine will show differences in change propagation for the various link types but similar results across all links.

7 RELATED WORK
Prior investigations of change propagation studied primarily the logical coupling between artifacts, i.e. which artifacts tend to co-evolve [11, 31, 32] and not the links among tasks. These approaches observe which artifacts frequently occur in the same commit (or in commits in temporal proximity) independent of the task these commits belong to. Zou et al. [34] apply interaction histories for detecting such coupling as they are more rich in information. They present a set of change patterns based on the temporal order of artifact reads and writes. Bantelay et al. [1] combine interaction histories and commit data from the Mylyn project to improve the detection of evolutionary coupling between artifacts. Kobayashi et al. [16] also use the interaction history to determine during an artifact change which other artifacts are accessed and what artifacts are changed in success. The resulting graph is used to predict which artifacts to change next. Robbes et al. [22] record detailed artifact changes from IDE interactions via SpyWare [21] for predicting sequential changes.

All these approaches scope their analysis to artifact changes within a single (often implicit) task. The underlying data sets either lack links among tasks or have no association of events to explicit tasks. Our data set enables for the first time the study of change propagation across linked tasks.

Several tools utilize developers’ interactions with the IDE to suggest relevant artifacts. NavTracks [26] supports navigation during software maintenance by suggesting related files based on the developers’ IDE interaction path in previous navigation sessions. TeamTracks [8] aims to ease program comprehension visualizing navigation patterns. Configurable HeatMaps [23] capture how often a file was changed or visited. Mylyn [15] is the most prominent tool that associates observed developer interactions with tasks. It thus determines the relevant artifacts for the developer’s underlying task. Similar, Hipikat [4] supports the developer in retrieving relevant artifacts from the project’s overall history. It considers documents, tasks, commits, messages, and artifact changes but not the detailed interaction history.

These tools offer assistance independent of tasks or focus on one single task context, i.e., the underlying conceptual models lack task links. We investigated how changes propagate across tasks.

Prior work studied work breakdown relationships based on task title [26] but didn't consider artifact changes to classify relations. Work on socio-technical congruence (STC) assesses team performance by investigating whether developers assigned to linked tasks also communicate and work on common artifacts. Initial work on STC [3] substituted co-evolving artifacts from commit data for explicit work dependencies. Valetto et al. [29], for example, propose mining software repositories to determine socio-technical congruence. Later approaches applied explicit tasks dependencies [13]. As our research has shown, developers use task links for managing diverse coordination needs, specifically that change propagation doesn't necessarily coincide with explicit links. A lack of artifact change propagation (or co-evolution), therefore, doesn't imply there is no dependency that needs managing and hence communication. Our work, thus, adds another challenge to measuring, understanding, and achieving social-technical congruence [24].

8 CONCLUSIONS AND OUTLOOK
We presented a quantitative and qualitative analysis of artifact change propagation in the Mylyn data set. We found seven situations describing how developers apply task links to manage implicit dependencies such as task synchronization and task continuation—not all of which are artifact-centric. This explains why change propagation occurs for only 64% of all linked task pairs. We identified additional six motifs that group non-linked task pairs according to either missing links or incidentally high change propagation.

We discussed the importance of our findings for development coordination support mechanisms. These rely on a classification of linked task pairs and non-linked task pairs. How exactly such mechanisms determine the respective situations and motifs automatically and reliably is subject to our future research. We intend to extend our qualitative analysis to all links in the Mylyn data set in order to identify complementary metrics based on graphs of directly linked tasks, artifact change frequency, and other similarity metrics (e.g., [30]) beyond change propagation.

ACKNOWLEDGMENTS
This work was supported by the Austrian Science Fund (FWF): P29415-NBL funded by the Government of Upper Austria.
5.6 Mining Cross-Task Artifact Dependencies from Developer Interactions

Authors: 
Usman Ashraf, Christoph Mayr-Dorn, Alexander Egyed

Venue: 
26th IEEE International Conference on Software Analysis, Evolution and Reengineering SANER (2019)
DOI: 
https://doi.org/10.1109/SANER.2019.8667990

Abstract: 
Implementing a change is a challenging task in complex, safety-critical, or long-living software systems. Developers need to identify which artifacts are affected to correctly and completely implement a change. Changes often require editing artifacts across the software system to the extent that several developers need to be involved. Crucially, a developer needs to know which artifacts under someone else’s control have impact on her work task and, in turn, how her changes cascade to other artifacts, again, under someone else’s control. These cross-task dependencies are especially important as they are a common cause of incomplete and incorrect change propagation and require explicit coordination. Along these lines the core research question in this paper is: how can we automatically detect cross-task dependencies and use them to assist the developer? We introduce an approach for mining such dependencies from past developer interactions with engineering artifacts as the basis for live recommending artifacts during change implementation. We show that our approach lists 67% of the correctly recommended artifacts within the top-10 results with real interaction data and tasks from the Mylyn project. The results demonstrate we are able to successfully find not only cross-task dependencies but also provide them to developers in a useful manner.
Mining Cross-Task Artifact Dependencies from Developer Interactions

Usman Ashraf, Christoph Mayr-Dorn, and Alexander Egyed

Institute for Software Systems Engineering
Johannes Kepler University
Linz, Austria

{usman.ashraf,christoph.mayr-dorn,alexander.egyed}@jku.at

Abstract—Implementing a change is a challenging task in complex, safety-critical, or long-living software systems. Developers need to identify which artifacts are affected to correctly and completely implement a change. Changes often require editing artifacts across the software system to the extent that several developers need to be involved. Crucially, a developer needs to know which artifacts under someone else’s control have impact on her work task and, in turn, how her changes cascade to other artifacts, again, under someone else’s control. These cross-task dependencies are especially important as they are a common cause of incomplete and incorrect change propagation and require explicit coordination. Along these lines the core research question in this paper is: how can we automatically detect cross-task dependencies and use them to assist the developer? We introduce an approach for mining such dependencies from past developer interactions with engineering artifacts as the basis for live recommending artifacts during change implementation. We show that our approach lists 67% of the correctly recommended artifacts within the top-10 results with real interaction data and tasks from the Mylyn project. The results demonstrate we are able to successfully find not only cross-task dependencies but also provide them to developers in a useful manner.

Index Terms—cross-task dependencies, change impact assessment, developer interactions, software artifacts recommendation, Mylyn, Bugzilla.

I. INTRODUCTION

Implementing a change such as fixing a bug, introducing a new feature, or removing outdated functionality is a challenging task. Correctly and completely implement a change requires the developers to identify all relevant artifacts. In non-trivial systems, a change often requires editing artifacts that are maintained by different developers or different teams. Aside from small, localized bug-fixes changes are rarely described and managed by a single task but rather a set of tasks worked on by different developers.

Crucially, a developer needs to know which artifacts under someone else’s control have impact on her underlying work task: changes to such artifacts then may induce additional changes, might restrict how to implement a change, and when to do so. In turn, a developer’s artifact changes cascade to other artifacts under someone else’s control. We characterize the situation when changes to artifacts in one task influence (potential) changes to other artifacts in another task as a cross-task dependency. Simple, illustrative examples include fixing a bug in business logic and updating integration tests accordingly, introducing a new field in the database and displaying it on the user interface, or introducing a feature toggle and adding the toggle trigger to the configuration database. From the authors’ experience, in many organizations often different teams are responsible for these tasks due to prescribed development processes, required expertise, or organizational structure.

Awareness of cross-task dependencies is especially important as lack thereof is a common cause of incomplete and incorrect change propagation. Developers thus need support in becoming aware of these dependencies, respectively the involved artifacts for simpler coordination of change propagation (i.e., forward and backward impact assessment [6]). Manually maintaining these dependencies is tedious.

Along these lines the core research question in this paper is: how can we automatically detect cross-task dependencies and use them to assist the developer?

We introduce an approach for mining cross-task dependencies from developer interactions with engineering artifacts as captured in the IDE. Interactions describe which artifacts a developer has accessed and edited within the scope of a task. From task pairs, we extract re-occurring artifact sequences: the cross-task dependencies. We further observe the accessed artifacts during live editing in the IDE and apply the dependencies to recommend which other artifacts are potentially affected by the ongoing work.

This paper provides the following two contributions: First, we developed a technique for extracting cross task dependencies from interaction data. To the best of our knowledge, this is the first attempt to obtain artifact dependencies across tasks. Contemporary approaches derive dependencies from within-task data or data from aggregated tasks of very close temporal proximity. Second, we provide a recommender prototype that applies developer interactions in order to suggest a ranked list of affected artifacts.

We obtained real developer interaction data and tasks from the Mylyn project and mined cross-task artifact dependencies in a sliding window over the duration of several years. Our approach lists 50% of correctly recommended artifacts within the top-5 results. The results demonstrate we are able to successfully find not only cross-task dependencies but also provide them to developers in a useful manner.
The paper is structured as follows. Section II provides a detailed motivating scenario to further outline the importance of this research followed by section III that outlines the technical details of our approach. Section IV then lists the specific research questions this paper investigates. Section V reports the study design with results available in Section VI and Section VII. We discuss the findings in Section VIII. Section IX compares our approach to state of the art before Section X completes the paper with conclusions and an outlook on future work.

II. MOTIVATING SCENARIO

We motivate our research based on a real example from the open source Mylyn project. Mylyn [15] allows a developer to connect to a task management tool (such as Bugzilla) for selecting tasks to work on and captures all developer read and write events within the Eclipse IDE. DevMM is assigned to new task 152211\(^1\) - Allow local tasks to be promoted to repository tasks, a feature in Mylyn’s Bugzilla sub component. DevMM obtains further details such as \textit{Provide facility to copy task from local repository to remote one} from the task description. Furthermore, he also examines comments from other developers on the same task which suggests their familiarity with the task. From the interaction history attached to the task we learn that DevMM browses through different artifacts; a usual activity to locate relevant source code artifacts.

Ideally, an impact analysis tool supports DevMM during this activity by suggesting potentially affected artifacts based on, for example, his browsing activities. In a developer’s experience, implementing a change often has impact on artifacts maintained by other developers. A developer will then assess to what extent his intended (or already implemented) changes will affect someone else’s code (forward impact assessment). That typically implies inspecting the potentially affected artifacts, and hence requires being aware of them in the first place. Similar, other developers’ artifacts (and their recent changes) may influence how a change can be implemented (backward impact assessment). Again, this involves being aware of such artifacts and inspecting them for relevance. Support is particularly important when these artifacts reside far away from the developers current work context, e.g., in a different (java) project or subcomponent.

In this particular example, our approach could support DevMM by recommending to inspect \textit{trac.core/.../internal/trac/core/TracTaskDataHandler.java}.\(^2\)

While task 152211 belongs to the Bugzilla subcomponent, this artifact belongs to the Trac subcomponent; a subcomponent primarily maintained by DevSP.\(^3\) DevMM would investigate, for example, whether the change for the Bugzilla subcomponent can be replicated in the Trac subcomponent.

Our approach is able to provide such a recommendation as it analyses past developer interactions and extracts which artifact dependencies exist across task boundaries. In this case, as soon as DevMM accesses \textit{.../tasks.ui/.../tasks.ui/editors/AbstractRepositoryTaskEditor.java} in the IDE we observe this event and search for matching cross-task dependencies. In the months prior to DevMM’s ongoing development efforts on task 152211, the Mylyn project exhibited six task pairs where \textit{...trac.core/.../internal/trac/core/TracTaskDataHandler.java} was accessed in one task and \textit{...tasks.ui/.../tasks.ui/editors/AbstractRepositoryTaskEditor.java} in another one. Listing 1 provides the corresponding cross-task artifact dependency excerpt as represented in JSON.

```
{
"id": "7ec6f597a31b6623e0207eeba23f199",
"miningStart": "2007-07-06",
"miningEnd": "2008-01-02",
"sourceArtifacts": [
  "org.eclipse.mylyn.tasks.ui/src/org/eclipse/
   mylyn/tasks/ui/editors/
   AbstractRepositoryTaskEditor.java",
  "org.eclipse.mylyn.trac.core/src/org/eclipse/
   mylyn/internal/trac/core/TracTaskDataHandler.java"
],
"destinationArtifacts": [
  "org.eclipse.mylyn.tasks.ui/src/org/eclipse/
   mylyn/tasks/ui/editors/
   AbstractRepositoryTaskEditor.java",
  "org.eclipse.mylyn.trac.core/src/org/eclipse/
   mylyn/internal/trac/core/TracTaskDataHandler.java"
],
"TaskPairs": [
  ["Id-196643", "Id-196622"],
  ["Id-196643", "Id-196585"],
  ["Id-196700", "Id-196622"],
  ["Id-196700", "Id-196585"],
  ["Id-196700", "Id-196643"],
  ["Id-196643", "Id-196700"
]
}
```

Listing 1: JSON representation of a cross-task artifact dependency

III. MINING AND APPLYING CROSS-TASK ARTIFACT DEPENDENCIES

Our approach to mining and applying cross-task artifact dependencies consists of three phases as depicted in Figure 1: (i) software development data gathering (1), (ii) dependency mining (2,3), and (iii) artifact recommendation (4,5,6,7). Artifact recommendation is one of many potential uses of cross-task artifact dependencies selected for demonstration purposes.

A. Software Development Observation

The basic data set from which we extract cross-task dependencies are developer interactions with engineering artifacts such as requirements, models, source code, documentation, test reports etc. (see Figure 1 (1)). In this paper, we focus on source code artifacts.\(^4\) A developer interaction describes how the developer interacted with the IDE such as opening, editing, closing artifacts, navigating the artifact structure, or executing commands. Various preexisting approaches such as Mylyn

\(^1\)https://bugs.eclipse.org/bugs/show_bug.cgi?id=152211
\(^2\)We abbreviated all artifact names in this section for sake of readability.
\(^3\)Bugzilla and Trac are two different bug tracking tools.
\(^4\)As the developer interactions from the evaluation system under study contain almost exclusively references to source code we cannot demonstrate that the approach would be successful in discovering cross-task dependencies among other artifact types.
capture such interactions. The key requirements for interaction data to be useful for our approach is providing following properties per interaction: who conducted the interaction (i.e., which developer), when did the interaction occur (we don’t need duration), what artifact (fragment) was accessed (read or write), and in the scope of which task (what bug report, issue, or story) did the interaction happen. Additionally, we require task details on assigned developers, task description, etc. As with interaction events, we store this information as task update events in the Unified Event Stream. We explain their use for measuring similarity among tasks in the following subsection.

We store all interaction events in the Unified Event Stream, thereby abstracting away which tool provided the interaction data. Ultimately, the Unified Event Stream contains the interaction events of all engineers working on the various tasks over the complete life-cycle of a software system.

**B. Cross-task Artifact Dependency Mining**

For dependency extraction, we decide on a time window (e.g., 6 months) and select tasks having been worked on actively during that interval (Figure 1 (2)). The window size depends on work intensity and the typical task durations. An overly wide window increases the risks of detecting primarily noise, i.e., false positive dependencies, while a too narrow window risks missing dependencies as it fails to include related tasks. We define a task as being actively worked on when a minimum number of interaction events have occurred (e.g., 10). Again, such a threshold depends on the typical number of artifacts accessed within the scope of a task. A high threshold implies that only tasks with a lot of events respectively involved artifacts will be considered for mining (thus potentially missing dependencies), while a too low threshold selects also task with little progress that are likely not part of a dependency (yet) but increase the mining effort.

In short, the preliminary input to the extraction algorithm consists of the active tasks, and for each task the set of artifacts that were accessed (read or write) within the time window according to the interaction events. Interaction events for an active task that occurred outside that time window are not considered.

The next challenge is generating meaningful task pairs from all active tasks in the observed time-window. Taking all possible permutations of tasks (i.e., the cartesian product) is not an option for two reasons: first, the number of pairs grows exponentially with the number of tasks and would require extensive computing resources for running the mining algorithm. Second, with all permutations the likelihood for finding noise increases. Our approach aims to identify task pairs that are more likely to exhibit cross-task dependencies.

Figure 2 depicts our task pair generation approach. Assume that out of tasks A to Z, querying the Unified Event Stream for active tasks in a given time-window t returns the tasks A, C, F, and X (1). We then calculate pair-wise similarity among tasks using properties such as common commenting developers or textual similarity in task description and comments (2). We use cosine similarity on the two sets of developers having commented on the respective tasks, which yields a similarity value of 0 for completely different commenting developers, and 1 for two tasks with the same set of commenting developers. With respect to textual similarity, we extract terms, apply stemming, and apply cosine similarity on the resulting term vectors. Overall similarity is simply a linear combination of these two metrics, giving each equal weight.

The rationale behind using developer and textual similarity as an indicator for cross-task dependencies is that the tasks are likely to share a common vocabulary (e.g., a database field update with impact on the user interface, very likely has the respective user interface artifacts use similar terms) and that relevant developers—even when not explicitly assigned to the task—share their expertise in comments. We deliberately ignore temporal distance among tasks as a similarity metric as, on the one hand, there are often large temporal gaps between related tasks, and on the other hand, developers try to work on unrelated tasks concurrently or in close temporal proximity as these require no or very little coordination.

From the resulting similarity matrix, we chose for each task the top-k most similar tasks (3) and collect them in a list. From each list, we generate the set of pairs (4). We then filter out
Fig. 2: Generation of TaskPairs.

all artifacts that occur insufficiently often (i.e., $< k \ast 2 + 1$) to become part of a pattern. Filtering artifacts may lead to an empty set, we then remove the respective sequence from the set. We abort mining once the number of sequences falls below the support threshold $s$. We introduce $s$ to obtain some initial confidence in a cross-task dependency. We store only those dependencies which the mining algorithm detects among at least $s$ distinct task pairs. The lower $s$ the more patterns we are likely to find, but also the more likely these represent noise. The higher $s$ the fewer patterns we can expect to find.

The actual problem of mining cross-task dependencies can be mapped to a simplified sequential pattern mining problem for which several algorithms exist [9]. Sequential pattern mining (SPM) takes a set of sequences, each sequence in turn consisting of item sets. SPM then searches for reoccurring patterns where one or more items in one item set are followed by items in a subsequent item set (within the same sequence). We use an algorithm by Fournier-Viger et al. [10] from the SPMF library.5

In our heuristic a sequence consists of a source tasks and a destination task (i.e., a task pair). The artifacts accessed within these tasks make up the respective item sets. We are interested which changes in one task (i.e., the source task) tend to be followed by (other) artifact writes and reads in another task (i.e., the destination task). Specifically, we include only write events in the source task, and include write and read events in the destination task. We do this to capture situations where a developer inspects the changed artifact to assess its relevance, respectively, where a change in one task results in artifacts automatically being (re)generated (e.g., model-to-code transformations) that are not changed again but rather read only.

Note that the temporal order of interaction events as well as the temporal order of tasks is irrelevant for dependency extraction as we assume cross-task dependencies to be time-insensitive. For example, it does not matter whether first one task updates a test, and then another task updates the corresponding functionality, or whether it is the other way around, or whether this happens simultaneously, the dependency exists nevertheless. We, thus, place every task pair twice into the set of sequences—once in temporal order, once in inverse temporal order—to detect dependencies independent of time. This has no negative effect on detecting dependencies that happen to be time-sensitive.

In our approach, we run the mining algorithm (5) at most $n$ times (once of each active task), which a maximum of $k \ast (k - 1)$ sequences per mining run—the mining effort thus grows linearly in $n$. In contrast, without generating task pairs, we would run the mining algorithm once but with $n \ast (n - 1)$ sequences. Listing 1 provides an example cross-task artifact dependency output. A dependency consists of the source artifact set, destination artifact set, the set of task pairs among which the mining algorithm detected the pattern, and the time-window. Specifically in Listing 1, editing AbstractRepositoryTaskEditor.java, coincided with accessing of the destination artifact TracTaskDataHandler.java among six task pairs.

After running the mining algorithm (6), we de-duplicate the found dependencies: for an identical pair of source and destination artifacts we simply merge the two task pair sets. Ultimately, we store the cross-task artifact dependencies for later use (7).

C. Artifact Recommendation

We expect cross-task artifact dependencies to be useful for multiple use cases. One such use case is supporting developers live during development by informing them about artifacts to inspect for impact on their underlying work (backward impact assessment) as well as what artifacts to inspect for impact of their work (forward impact assessment). Our recommendation prototype listens to artifact read and write events during a developer’s work in the IDE (i.e., the same type of events as stored in the Unified Event Stream, Figure 1 (4)).

For every few events, the recommender checks if it has used the accessed artifact as a trigger for a recommendation before. If this is not the case, it queries the cross-task artifact dependency database for any dependency that lists the artifact among the source artifacts (5). From all matching dependencies, the recommender then retrieves all artifacts from the destination set, filtering out those that have been recommended

5 http://www.philippe-fournier-viger.com/spmf
Once before. We recommend each artifact at most once to avoid annoying the developer. The recommender then ranks the remaining artifact candidates along the following criteria to rank the most relevant ones first.

We apply the following criteria (6):

**Occurrence** counts across all matched dependencies in how many tasks the candidate artifact was accessed. We assume a higher count implies more relevance.

**Distance** measures the package hops from the triggering artifact to the candidate. Similar to [7], we count how many packages up the hierarchy and down again does a developer has to navigate to reach one artifact from the other. For example, the artifact `org/eclipse/tasks/Class1.java` is three hops away from `org/eclipse/internal/sandbox/Class2.java` as we need to traverse tasks up and internal/sandbox down to reach `Class2.java` from `Class1.java`. We assume a more distant candidate artifact to be more relevant than a closer one as a developer may be less aware of the change impact than on an artifact that is close to his/her current work context.

**Access Frequency** counts the number of tasks (in the time-window used for mining dependencies) in which any developer accessed the candidate artifact. In contrast to Occurrence, this metric evaluates the popularity of an artifact. We assume, a frequently changed artifact has more impact than an infrequently changed one.

**Personal Access Count** determines in how many tasks the currently active developer (the one about to receive a recommendation) has accessed the artifact candidate before.

We normalize each score to the range $[0, 1]$ for comparability and multiplying each score by its respective weight, where $w_i \in [0, 1]$ and $\sum_i w_i = 1$. The sum of weighted scores produces the overall score used for ranking all artifact candidates. The recommender returns the ranked list of artifacts (i.e., a recommendation instance, Figure 1(7)) and internally stores for each recommended artifact the ranking metric results and applied ranking weights. This enables dynamically adapting the weights upon observing what artifact the developer eventually accesses. Such analysis and self-tuning, however, is out of scope of this paper.

**IV. Research Questions**

In our paper, we split the research questions (RQ) into two coarse grained groups: (1) what are the characteristics of detected cross-task dependencies? and (2) are the detected cross-task dependencies indeed useful?

**RQ1a:** Do we find cross-task dependencies with our proposed heuristic? Answering this question informs us whether we are able to detect reoccurring development situations where accessing an artifact in one task (i.e., the source artifacts) tends to coincide with accesses to an artifact in another task (i.e., the destination artifacts). At this stage, however, we remain unsure whether the found relations are true dependencies or just noise.

**RQ1b:** How much are the source and destination artifacts overlapping. Put inversely: to what extent are artifacts in the source set disjoint from artifacts in the destination set?

A large overlap would imply that the same artifact are changed together in multiple tasks and hence that contemporary approaches based on logical coupling, for example, would be equally able to find these dependencies.

**RQ1c:** What is the ratio of unique artifacts in the dependencies compared to all unique artifacts?

A high ratio might indicate that our heuristic finds a lot of noise as we wouldn’t expect the majority of artifacts to be part of a cross-task dependency. A high ratio points to poor cohesion and tight coupling of artifacts (and teams!) across the system.

**RQ1d:** Are the task pairs—among which we find task dependencies—linked in the issue tracking system?

If so, then our heuristic could focus on analysing linked task pairs only. Previous work has shown that linked task pairs not necessarily exhibit access of the same artifacts [20]. It didn’t investigate, however, the presence of cross-task dependencies.

**RQ2a:** Are we able to predict based on cross-task dependencies whether developers need to become aware of a particular artifact given their current work task context?

If so, the task dependencies can be applied in a recommender to inform a developer about artifacts that might have been changed in another task context before, respectively what artifacts might have to be changed by another developer, thereby assisting forward and backward change propagation. If we can’t predict artifacts then the possibility exists that our heuristic detects mere noise in the interaction data.

**RQ2b:** Are we able to predict artifacts in a meaningful manner?

If we are able to predict artifacts but these are buried among a large number of false positives then the developer is burdened with evaluating the correctness of the recommendation and—over time—will likely ignore the recommendations. Thus we achieve little benefit.

**V. Study Design**

**A. Data Gathering**

We use developer interactions captured via the Eclipse IDE Mylyn Plugin\(^6\) during the development of the Mylyn project [15]. Mylyn tracks the development context of an interaction which provides us with details on who was working on which bug report, the type of interaction (e.g., selection, write, navigation) and the timestamp of the interaction. We have public access to these interaction events in the form of attachments to bug reports on the Eclipse Bugzilla website.\(^7\) We interpret a bug report as a task in our approach. We retrieved all bug reports including their change history, comments, and interaction attachments from the Mylyn project that listed at least one

---

\(^6\)http://www.eclipse.org/mylyn/

\(^7\)https://www.bugzilla.eclipse.org/bugs/query.cgi
Mylyn interaction attachment via Bugzilla’s JSONRPC API. For each interaction attachment, we extracted all read events (i.e., selection) and write events (i.e., edit) at the file level and stored them in the Unified Event Stream (ignoring all other events as they don’t impact this work). We additionally added events for each task change and each comment as needed for task similarity calculation. In total, we processed 4,477 tasks with interaction attachments, resulting in over 417,000 events that cover the interval from 15th November 2005 to 4th April 2017. We observed the majority of interaction events in the interval between 2006 and 2013.

B. Dependency Mining

We applied a sliding window approach for mining dependencies and then evaluating their usefulness.

Within a training window, we selected all tasks with at least 10 artifact access events in the Unified Event Stream as the training tasks. This selects only those tasks that have their main work effort fall into the training window. We generated task pairs with \( k = 4 \) as outlined in Section 1 and executed the mining algorithm with a support threshold of \( s = 5 \), a trade-off between computation time and likelihood to detect a dependency.

Inspecting the Mylyn data set, we noticed that related tasks (i.e., tasks that likely exhibit cross-task dependencies) often have significant temporal gaps due to the nature of open source development. We therefore determined a 180 day training window to be large enough to include related tasks and small enough to detect dependency changes across time.

C. Dependency Evaluation

We take the time (i.e., evaluation) window immediately following the training window for evaluating the detected dependencies. We chose a 30 day evaluation window size as this selects tasks that are not too far away from the mining window and long enough to select the majority of work going on in the task. Within the evaluation window, we selected all tasks with at least 10 artifact access events in the unified event stream as the evaluation tasks. This selects tasks for evaluation only during intervals of their main work effort; not at the task’s beginning when there are too few events to use as input for recommendation, nor towards the task’s end when recommendations wont be useful.

We simulated developer work on a particular task by replaying all interaction events for that task within the evaluation window in temporal order. For every five events—the first five are the seed—we trigger the recommender with replayed events and obtain its artifact recommendations (see Figure 3 (1)). Hence, we obtain for each task a set of recommendation instances, each containing a list of one or more ranked artifacts (4).

For each recommended artifact, we mark it as a true positive when the interaction event sequence yet-to-be-replayed contains an event accessing that particular artifact. We mark the artifact recommendation as a false positive when no such event occurred. Given that we recommend multiple times, we further record its recommendation-specific rank (i.e., its position within the current set of artifact recommendations), and its task-specific rank (i.e., its position among all recommendations for a task).

At the end of each task replay, we mark all artifacts as false positives which appeared in the Unified Event Stream in any prior training interval and that were not part of the seed (7). We excluded any new artifacts from the false negative set as we cannot predict an artifact we haven’t encountered before.

After mining dependencies from the 180 day window, and evaluating them in the subsequent 30 day evaluation window, we would then shift the training and evaluation window by 30 days and repeat the procedure. Given the lack of active tasks (i.e., insufficient developer interactions per task) during the first few months, we set the first training window to 12th May 2006 to 8th November 2006. Similar, we set the end of the last training window to 6th May 2013 for a total of 80 training, respectively evaluation, iterations. Events after March 2013 were either too few to consider a task as active, or consisted merely of task updates and comments.

The following section reports results from the quantitative analysis.

VI. DEPENDENCY MINING QUANTITATIVE ANALYSIS

This section reports on the quantitative analysis results required for answering research questions RQ1a-d.

Figure 4 displays dependency statistics for 60 training window iterations starting May 2006 to Nov 2011. We select between 2 and 6 tasks for mining until May 2013. These are, however, too few for our mining algorithms to find patterns and hence we don’t display iterations 61 to 80. Our Supporting Online Material includes the data set with all iterations, though.

Answering RQ1a: Do we find cross-task dependencies with our proposed heuristic?

Figure 4 (top) reports how many unique artifacts developers accessed in events across all tasks within each iteration (\( \forall \)), respectively how many unique artifacts exist in the code base from the first iteration until the end of the nth iteration (\( \Delta \)).

Figure 4 (middle) visualizes the number of training tasks, the amount of unique dependencies found the number of unique artifacts across all source sets, respectively destination sets; again based on events from a single iteration.

Our approach detects the first dependencies in iteration 3, the last ones in iteration 50, with a total of 1,813 dependencies. In iteration 13 we find the most dependencies: 235 dependencies, respectively 372 normalized dependencies, made up of 72 unique artifacts among the source artifacts, respectively 124 unique artifacts among the destination artifacts.

Overall, these results show that our mining heuristic finds a significant number of cross-task artifact dependencies. The
bulk of these dependencies occur during the initial growth phase of the project. We note a sudden spike in unique artifacts in iteration 17 which sees a sudden increase of almost 5,000 new artifacts. This spike interrupts the gradual slowdown of newly introduced artifacts which we observe before and after that iteration. We assume that this is the result of a major restructuring of the code base and/or integration of sub components previously not covered in the interaction traces. Our approach would require artifact, respectively source code commit, inspection capabilities to detect artifact renaming, respectively, artifact relocation to another package. At the moment, our approach treats the artifact before and after renaming as two distinct artifacts. Consequently, our mining heuristic is less likely to find dependencies involving the renamed artifacts, and existing dependencies are no longer relevant during recommendation and result in false positives. This is one explanation why the number of detected dependencies is low after iteration 17 (see Figure 4 middle) even though the number of unique artifacts accessed per iteration remains high (see Figure 4 top).

Up to iteration 8 and beyond iteration 25, a lack of accessed artifacts explains the low number of detected dependencies.

Answering RQ1b: How much are the source and destination artifacts overlapping? [...] Figure 4 (bottom, red $x$) reports the median Jaccard similarity coefficient (i.e., set intersection over union), measuring the overlap of artifacts in the source set and destination set for detected dependencies. A Jaccard coefficient close to 1 describes sets with almost identical members, while a value close to 0 describes two sets that share virtually no members.

The median Jaccard coefficient rarely rises above zero. Only at later intervals where we hardly find dependencies (compare with Figure 4 middle - Dependencies Detected) do we experience non-negligible overlap. This implies that for at least half of all dependencies there is no overlap between source and destination artifacts. This is a strong indicator that we are able to find dependencies that cannot be detected with logical coupling techniques. Logical coupling requires the coupled artifacts to repeatedly appear together in the same commit. If cross-task dependencies were just logically coupled artifacts, then we would find similar source and destination sets (i.e., the same artifacts appearing in source and destination) consequently yielding high Jaccard similarity. Figure 4 (bottom, red $x$) shows that this is not the case.

Answering RQ1c: What is the ratio of unique artifacts in the dependencies compared to all unique artifacts? Figure 4 (bottom, green $□$ and blue $○$) depicts the ratio of artifacts in the source set, respectively destination set, with respect to all accessed artifacts in that iteration.

There we observe how only for a single iteration in the beginning of the project the source artifacts consist of ~13% of all accessed artifacts. This number quickly drops to single digits, and eventually fluctuates around 1%. Destination artifacts show an even earlier drop and remain similarly low. These numbers indicate that only a handful of artifacts make up a cross-task artifact dependency, and hence explain why recall is not a suitable metric for evaluating the benefit of our approach.

Answering RQ1d: Are the task pairs—among which we find task dependencies—linked in the issue tracking system? Explicit links between task pairs of a dependency (e.g., Bugzilla’s blocks and depends on relations) occurred extremely rarely (hence not shown). In total, we found only in 13 out of 1,813 dependencies where at least one task pair exhibited an explicit link. We encountered these dependencies in iterations 10, 11, 12, 13, 17, and 30.

Observing an explicit link among task pairs in less than 1% of all dependencies is a strong signal that explicit links cannot be used as a filter/predictor among which task pairs we are likely to find a cross-task artifact dependency.

VII. Recommendation Quantitative Analysis

Figure 5 (top) displays for each iteration the number of tasks that received no recommendation ($□$), no correct recommendation ($●$), and tasks with at least one correct recommendation ($○$). Evaluation iteration $i$ identifies the 30-day window following the $i$th training iteration.

Over the 50 iterations, we replayed a total of 2,086 evaluation tasks. We stop at iteration 50, the last iteration where we detected dependencies.

Answering RQ2a: Are we able to predict based on cross-task dependencies whether developers need to become aware
of a particular artifact given their current work task context? We provide recommendations (i.e., recommend at least one artifact) for 798 out of the 2,086 evaluation tasks, a task coverage rate of 38%. Out of these, 229 contain one or more correct recommendations. On the task level, we thus achieve a precision rate of 29%.

A task coverage rate of 38% and task precision rate of 29% is expectedly low as a well-designed software system enables the majority of tasks to be worked on independently, i.e., without affecting other tasks and hence involving no artifacts which are part of a cross-task dependency. This fact is also apparent in low recall values (see Figure 5 bottom, ◦).

Low coverage and recall is not an issue as our goal is NOT to recommend every artifact a developer should access but rather aim to focus recommendations to those that have potential cross-task impact and thus might need dedicated coordination.

Answering RQ2b: Are we able to predict artifacts in a meaningful manner?

We compute the mean reciprocal rank (MMR) using task-specific rank (TSR) across tasks for every interval and report in Figure 5 (bottom): once for all tasks with a recommendation (+) and once only for tasks with at least one correct recommendation (×). The overall MMR for tasks with at least one correct artifact recommendation is 0.60; and 0.18 across all tasks.

Out of the 229 tasks, we are able to provide in 214 instances at least one recommendation instance that has a correct artifact within the top 10 results (i.e., 93%). Overall, the median recommendation instance-specific rank (RISR) of a correct artifact recommendation (i.e., independent of tasks) is 5, with 67% of artifacts having an RISR <= 10. This implies that in 67% of recommendation instances, whenever there is at least one correctly recommended artifact, the developer will find it within the top 10 results. Hence, a developer needs not look far down the list of artifact recommendations to obtain at least one useful recommendation. This is in range of the typical search distance when browsing search engine results on the web.

VIII. RESULT DISCUSSION

In this section, we discuss the implications and limitations of our approach and its evaluation.

A. Implications

Overall, the results paint a promising picture that developer interaction events are suitable for detecting cross-task artifact dependencies. We refrain from the claim, however, that such events should be the only input to dependency detection. The quantitative evaluation of recommendations highlighted that additional efforts need to be put into determining when a cross-task dependency is relevant as the precision rates leave room
for improvement. Determining dependencies based on artifact access events below file level might allow for more accurate dependencies and recommendations.

While live recommendations to developers are one way to make use of the detected dependencies, potential alternative uses include providing insights during explicit impact assessment activities, code reviews, task scheduling, or software system architecture inspection. Our results motivate the evaluation of these application scenarios as part of future work.

B. Limitations
We evaluated our approach with artifacts at the file level as we hypothesize that cross-task dependencies among Java classes are likely to exist at the file level rather than at the method level or below. While our approach is generic enough to work on any artifact granularity level, extra effort and evaluation is needed to confirm dependencies detection among, for example, model elements in UML and source code, or among source code and documentation.

Another limitation is support in the presence of newly created artifacts. We can only provide recommendations for artifacts that developers accessed in previous tasks that were subsequently mined and hence might not be aware of the most recent cross task dependencies. In this respect our approach has the same limitations with regard to new artifacts as logical coupling techniques. We argue, however, that we generate recommendations as soon as the developer accesses an existing artifact (e.g., to check how something was implemented so far). Additionally, we don’t expect an immediate impact when only new artifacts are introduced.

C. Threats to validity

Internal Validity We address researcher bias by analysing data from an open source system rather than conducting controlled experiments. The analysis focused on artifacts and tasks and was not specifically tailored to Java development in general or the Mylyn project in particular. With respect to the data set quality, we noticed that not all tasks in the Bugzilla issue tracking system provided an interaction attachment. As Mylyn interaction attachment upload is neither automated nor mandatory, we were limited to a subset of all tasks. Hence, we very likely were unable to detect several dependencies, respectively couldn’t evaluate them on tasks that might have benefitted from them. Overall, the Mylyn dataset is rich enough for a sufficiently long, continuous interval to allow successful mining and recommendation.

External Validity We analysed only a single data set as we are not aware of other real world projects aside from Mylyn that make a significant amount of task-centric interaction events available. Mylyn interaction data upload capabilities are not available by default and thus not widely used beyond the Mylyn project. Hence, we are careful to generalize our findings beyond the scope of the Mylyn project. Our analysis, however, demonstrated that it is indeed possible to detect cross-task dependencies from interactions and motivates further research in this direction. As outlined in future work (Section X), alternative data sources such as commit information might possibly allow for cross-task mining. Commit data, however, lacks (i) read-only events which reduces the detection rate and (ii) temporal information which precludes interaction replaying for evaluation purposes.

Construct Validity The replay approach is a proxy of usefulness as we consider only those recommendations as successful where the developer eventually accessed the recommended artifact. We might have recommended artifacts that are indeed relevant but the developer at that time wasn’t aware of these artifacts, hence didn’t access them, and we therefore regarded them as false positives. We, therefore, cannot assume that all false positives were indeed inaccurate. We thus can primarily claim that our recommender is helpful for remembering which artifacts to assess, which in non-trivial systems is important nevertheless. We refrained from interviewing developers from the Mylyn project as the time frame with sufficient interaction data to detect dependencies and evaluate them is almost

![Fig. 5: Tasks with correct/without correct/without any recommendations (top); Recommendation Recall and MRR (bottom).](image-url)
10 years ago. We would not expect feedback to provide meaningful insights after such a long time.

IX. RELATED WORK

Investigations into change impact among code artifacts studied the logical coupling between artifacts, i.e., which artifacts tend to co-evolve [27], [29], [30]. These approaches observe which artifacts frequently occur in the same commit (or in temporal proximity) independent of the task that the changed happened in.

Few approaches analyse control and data flow among code artifacts [25], mine association rules from software revision histories [17], [23], or utilize a variability model to detect the impact within product families [1].

Several researchers consider developers’ interaction histories to augment traces among logically coupled source code artifacts [2], [3], [16]. Kostadin et al. [5] use low-level IDE interaction to detect hidden behavior of developers. Bantelay et al. [2] combine interaction histories and commit data to improve the detection of evolutionary coupling between artifacts. Sebastian et al. [22] use developer activities in the IDE with context information, such as source-code snapshots for change events to study developer behavior. These approaches aim to find traces among code artifacts without considering contextual information such as the task the developer is working on. However, Wiese et al. [28] apply contextual information collected from tasks, developers’ communication, and commit data to capture the change patterns of artifacts. They use this contextual information to improve the artifacts co-change prediction. We focus specifically on cross-task dependencies for identifying dependencies with dedicated coordination needs.

Several task-centric approaches consider fine-grained developer interactions but restricts analysis to interactions within a task without considering the relations to other tasks. Kersten and Murphy [15] introduce the Mylyn tool for determining which code artifacts are relevant for a particular development task. Their analysis, however, is limited to a single developer within a single task.

Hipikat [4] supports the developer in retrieving relevant artifacts from the project’s overall history. It considers documents, tasks, commits, messages, and artifact changes but not the detailed engineering interaction history. A tool that can capture the interaction occurred in a particular file is HeatMap [24]. Another tool, Wolf [8] extracts artifact ownership and changes from source code repositories and generates traces between artifacts and engineers. The tool provides an organizational view for managers and an individual view for developers to support impact analysis activities. However, such tools focus on the relation between artifacts and tasks but not necessarily on the dependencies of artifact across tasks.

Multiple authors investigate inter-task relationships. Thompson et al. [26] study how software developers use relationship between tasks based on their titles to breakdown project work. Their study indicates that finding relationship between tasks can improve software development techniques. Mayr-Dorn et al. [20] investigated if the propagation of artifact changes across tasks reflect work dependencies among them. They observed that task links are useful for recommending artifacts to monitor for changes and these links can also potentially be used to recommend cross task dependencies. However, their focus was on whether the same artifact is changed in two tasks, while we investigate whether distinct artifacts are changed.

Related work with respect to horizontal traces (i.e., dependencies among code and non-code artifacts) falls into two categories: (semi)-auto-matically establishing traces [12] and maintaining traces under system evolution. Examples include Guo et al. [13] who apply domain specific knowledge to generate traces between requirements and code. Ghabi and Eyed [11] identify likely incorrect or missing traces between requirement and code by comparing trace patterns and source code calling relationships. Mahmoud and Niu [19] suggest refactoring techniques to improve and re-establish traceability between requirement documents and source code.

Examples for trace maintenance approaches include Mäder et al. [18] who use UML model changes to trigger automatic traceability maintenance rules. Jiang et al. [14] apply incremental latent semantic indexing to automatically manage traceability links between code and documentation. Nejati et al. [21] demonstrate the use of natural language processing to automatically identify the impact of requirements changes on system design.

Approaches to supporting the management of horizontal trace represent orthogonal approaches to our work. Combining horizontal traces approaches with cross-task artifact mining could potentially identify which horizontal traces require the most coordination effort. Alternatively, our approach could identify implicit horizontal traces that have not been explicitly modeled.

X. CONCLUSIONS AND FUTURE WORK

We presented an approach for mining cross-task artifact dependencies from developer interaction events. We described a heuristic for pairing up tasks from which to mine dependencies using a state-of-the-art sequential rule mining algorithm. Detected dependencies allow the recommendation of artifacts to be inspected for change impact analysis. We evaluated our approach on the Mylyn data set and demonstrated that we are able to detect dependencies over a considerable project duration (given the available data) that also resulted in usable recommendations to the developers. We provided correct recommendations in ∼30% of all tasks where we had matching dependencies and ranked 50% of all artifact recommendations within the top 5 results.

Our future work consists of two activities. On the one hand, we will evaluate the impact of excluding read-only data and compare the results obtained from the Mylyn data set with other data sets that consist of commit data only. Publicly accessible issue trackers such as Eclipse Bugzilla, Apache Jira, and Github host various projects that provide task and commit data. On the other hand, we will focus on evaluating other uses of dependencies such as task dependency analysis, task similarity analysis, or developer network analysis.
REFERENCES


5.7 Do Communities in Developer Interaction Networks align with Subsystem Developer Teams? An Empirical Study of Open Source Systems

Authors:
Usman Ashraf, Christoph Mayr-Dorn, Atif Mashkoor, Alexander Egyed, and Sebastiano Panichella

Venue:

DOI:
https://doi.org/10.1109/ICSSP-ICGSE52873.2021.00016

Abstract:
Studies over the past decade demonstrated that developers contributing to open source software systems tend to self-organize in “emerging” communities. This latent community structure has a significant impact on software quality. While several approaches address the analysis of developer interaction networks, the question of whether these emerging communities align with the developer teams working on various subsystems remains unanswered. Work on socio-technical congruence implies that people that work on the same task or artifact need to coordinate and thus communicate, potentially forming stronger interaction ties. Our empirical study of 10 open source projects revealed that developer communities change considerably across a project’s lifetime (hence implying that relevant relations between developers change) and that their alignment with subsystem developer teams is mostly low. However, subsystems teams tend to remain more stable. These insights are useful for practitioners and researchers to better understand developer interaction structure of open source systems.
Do Communities in Developer Interaction Networks align with Subsystem Developer Teams? An Empirical Study of Open Source Systems

Usman Ashraf  
Christoph Mayr-Dorn  
Atif Mashkoor  
Alexander Egyed  
Johannes Kepler University, Austria

Sebastiano Panichella  
Zurich University of Applied Science, Switzerland

Abstract—Studies over the past decade demonstrated that developers contributing to open source software systems tend to self-organize in “emerging” communities. This latent community structure has a significant impact on software quality. While several approaches address the analysis of developer interaction networks, the question of whether these emerging communities align with the developer teams working on various subsystems remains unanswered. Work on socio-technical congruence implies that people that work on the same task or artifact need to coordinate and thus communicate, potentially forming stronger interaction ties. Our empirical study of 10 open source projects revealed that developer communities change considerably across a project’s lifetime (hence implying that relevant relations between developers change) and that their alignment with subsystem developer teams is mostly low. However, subsystems teams tend to remain more stable. These insights are useful for practitioners and researchers to better understand developer interaction structure of open source systems.

Index Terms—developer interaction network, system modularity, subsystem coordination, developer communities.

I. INTRODUCTION

Global software development is carried out by developers located in various parts of the globe. They face a number of challenges in relation to communication and coordination due to the distances involved in three dimensions – geographical, temporal, and socio-cultural [1]. Open source software development is considered as a successful example of large scale global software development [2]. In open source development environment, software systems do not have a pre-assigned organizational structure as developers can contribute to any part of the system. However, developer communities are self-organized within development teams [3]. Studies have shown that the developer communities organization has a significant impact on software quality [4]–[7].

Motivated by the aforementioned insights, researchers have built automated tools analyzing developers interaction data from various sources, such as version control systems, mailing lists, or issue trackers [3], [8]–[12], to investigate how “emerging development teams” are formed in open source projects.

However, a few researchers observed how the emerging “community structure” aligns with “Subsystem developer teams”. Bird et al. [3], for example, hypothesized that files edited by developers of the same community are placed “closer together” than the files edited by randomly picked developers. Results, however, were inconclusive.

We argue that observing the alignment of developer communities with the system architecture can be more accurately studied at the level of coarse-granular subsystems, i.e., the components of a system, rather than analyzing these aspects at the level of fine-grained artifacts. In this regard, the work of Lenarduzzi et al. [13], for example, is important as they point out the potential negative impact of team independence at the subsystem (microservice) level. Additionally, Nagappan et al. [4] have shown that organizational metrics are the top bug proneness predictor in an industrial setting. Given the lack of explicit organizational structures in open source systems, we suggest to utilize implicit structures (i.e., communities)1.

Take the excerpt from a fictive project in Fig. 1 as an example. The developer interaction network on the top exhibits edges based on being involved in the same issue via activities such as reviewing or commenting. A community detection algorithm assigns developers to communities (blue borders) based on the developers’ interaction intensity (indicated by line thickness). Developers who interact often tend to form communities. The lower part of Fig. 1 depicts files and issues belonging to three subsystems A, B, and C. Subsystem team members are then those developers that changed an artifact belonging to that subsystem, respectively, are active in an issue referring to this subsystem. Note that subsystem member sets may overlap as in open source projects key developers are often involved in multiple subsystems. In our example, developer Dave is a member of all three subsystems, Carol is a member of subsystem A only. In this example, when comparing the members of subsystem B (i.e., Dave and John),

1We define a community as the set of developers that share responsibility or interest such as working on the same subsystem(s) [12].
we notice that they are members of different communities and there is no edge between them. This could be a warning signal about poor communication and coordination between the two.

To shed light into how communities and subsystems are aligned, we empirically investigate in this paper, in the context of 10 open source projects, how communities emerge and change over time – e.g., how developers join and leave sub-communities – and the extent to which these community patterns match the subsystems evolution. We find that developer communities change considerably across a project’s lifetime (hence implying that relevant relations between developers change) while subsystem developer teams (SDTs) remain comparatively stable. Overall, the community alignment with SDTs is often low, which implies that developers maintain significant communication ties with developers outside their (subsystem) work scope. We hypothesise that such an interaction network independent from subsystems emerges from the need to remain robust against the disruption of leaving developers and quick onboarding of new members.

The primary contribution of this paper is an empirical study investigating the evolution of developer communities and their alignment with the subsystems developer teams (Section IV). The secondary contributions of this paper are: i) a technique for measuring alignment (i.e., overlap) between subsystems and developer communities (Section IV(C)), and ii) a technique for determining developer communities evolution (Section IV(D)).

The paper is structured as follows. Section II introduces the research questions. Section III describes the applied data gathering method and the resulting evaluation data set. Section IV explains our community-SDT alignment and evolution measurement technique with Section V presenting the results. We discuss the obtained results and their implications in Section VI. Section VII compares this study to state of the art approaches before Section VIII, which concludes the paper with a summary and an outlook on the future work.

II. STUDY DESIGN

Developers that work on the same task, artifact, or subsystem need to coordinate and thus communicate in order to avoid incomplete change propagation, rework, or duplicate work. We therefore would expect them to form stronger interaction ties [14] than developers working in different subsystems. We hypothesize that these developer interactions give rise to a community structure. Therefore, the goal of this study is to obtain first insights into the alignment of sub communities within open source projects with the systems’ structure. The motivation behind this study is to determine the extent to which emerging open source developer communities reflect the subsystem structure, proposing mechanisms for studying such communities aspects. The perspective is of practitioners and researchers that could leverage such mechanisms to identify subsystems where developers coordinate insufficiently.

The context of this study comprises commits, issues, source code (folder) structure, and the conceptual and structural links among the data. The selected 10 open source projects are part of our publicly available and published dataset [15].

Research Questions:
In this work, we investigate the following research questions:

RQ1: To what extent can we identify well defined developer communities across the investigated projects’ lifetime? Our goal is to propose a systematic approach able to detect emerging communities by analyzing the developer interaction network generated from issue involvement and commits linked to those issues (see Fig. 1). We are then interested in knowing how often and how many well defined communities we detect.

RQ2: Do the developers active in the same subsystem emerge in the same development communities? In other words, measuring the overlap of the SDTs with the detected communities allows us to identify subsystems where the communication among developers occurs mostly within the subsystem.

RQ3: How stable are the detected communities across time compared to the SDTs? While RQ1 and RQ2 investigate developer interaction network and its alignment with SDTs at separate snapshots in time, here, we inspect the stability of the detected communities and the SDTs across time. We analyse whether a pair of developers that belong to the same community in one time window also belong to it in the subsequent time window. Likewise, we are interested to know whether a pair of SDTs that emerge in the same developer community does so in the subsequent time window.

Overall, answering these questions provides insights into whether subsystems in open source projects represent decoupled work scopes that result only in limited coordination overhead compared to work coordination within a subsystem (as measured by interaction ties).

III. DATA GATHERING AND PREPROCESSING

A. Open Source Project Selection

The project selection (Fig. 2 (1)) is the first step of our study approach. We select the candidate project according to the following requirements:

Fig. 1. Example developer community to subsystem alignment.
**Subsystem Structure** We manually selected projects which exhibit a non-trivial (i.e., at least 10 subsystems) and clear subsystem structure to avoid introducing potential bias by splitting the system into subsystems that do not reflect the real underlying decomposition.

We manually inspected the projects on Apache’s Jira server\(^2\) that exhibited a significant number of issues with a component property set and where the top level source code folders (hosted on Github) closely (or identically) matched those component names. We interpreted those components/folders as subsystems.

We avoid selecting projects from a single mechanism for structuring subsystems by also selecting Github projects that manage code across multiple repositories (multi-repo), i.e., one repository per subsystem, as our selected Jira projects are mono-repo, i.e., one folder per subsystem.

**Developers** Further selection criteria included a minimum of 40 participating developers over the project’s life time to guarantee that even with heavy fluctuation of developers, that there are sufficiently many developers to form an interaction network with sub-communities. Note that for our selected projects the number of developers are well over 40.

**Commits and Issues** Furthermore, we filtered out projects with less than 1500 commits or 200 issues. This ensures that a relation between two developers based on being active in the same issue or committing an artifact update linked to the same issue may occur sufficiently often to signify a meaningful relation between the developers and not just an one-off occurrence.

This study focuses on 10 projects listed in Table I. The projects are limited in number and size as (i) there is a manual processing effort required in step 1 (Fig. 2) and (ii) non-negligible manual effort is necessary to investigate folder structure and confirm developer deduplication.

---

\(^2\)https://issues.apache.org/jira/

---

**B. Data Extraction and Storing**

Having identified 10 projects under investigation, we extracted commit and issue information (Fig. 2 (2)). We used Perceval [16] to extract commit and issue information from Github and the Jira python client (Perceval lacks to provide our desired Jira issue information) for extracting data from Apache Jira. Both these tools provide data in the JSON format.

Commits, the involved artifacts, and issue details make up the core information of our data set. In our datamodel, an issue may represent a Jira issue, a Github issue, or a Github pull request, or even a combination thereof whenever, for example, a pull request references a Jira issue.

A developer participates in a project in various ways: for example, from committing code changes, to commenting on issues, to reviewing pull requests. We introduce two *Involvement* types to harmonize activities across Jira and Github.

*Contributing* is equal to committing artifacts (i.e., as indicated by a commit linked to an issue); *Informative* describes input to an issue such as having reported it, commented on it, or having reviewed artifacts. We encode these two types of actions as integers of value 3 and 2, respectively, to reflect the amount of effort behind the activities. Before continuing with these values we performed a value sensitivity analysis by assigning them values (4,2) and (1,1) respectively. We observed little change in the detected community structure with relatively less quality. More details of community structure and its quality metric are in the Section IV(A).

**C. Developer De-Duplication**

When recording a developer’s involvement across commits and issues, we need to take care of situations where a developer uses multiple accounts (email-ids) on Github. More importantly, a developer’s id on Jira does not match the Github account id of the same developer as Jira does not use email addresses as a part of a user id while Github does. This will result in inconsistencies as one developer will appear...
TABLE I
OVERVIEW OF THE TEN ANALYSED PROJECTS.

<table>
<thead>
<tr>
<th>Project</th>
<th>Type</th>
<th>Programming Language(s)</th>
<th>Time Period</th>
<th>Commits</th>
<th>Devs</th>
<th>Subsys</th>
<th>Arts</th>
<th>Issues</th>
<th>Art(s) Linked To Subsys (%)</th>
<th>Issues Linked To Subsys (%)</th>
<th>Commits Linked To Subsys (%)</th>
<th>Commits Linked To Issues (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>apache</td>
<td>Mono-Repo</td>
<td>Scala, Java</td>
<td>11 (Oct 10-Nov 11)</td>
<td>10,689</td>
<td>364</td>
<td>17</td>
<td>2,406</td>
<td>1,000</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>nameko</td>
<td>Mono-Repo</td>
<td>Python</td>
<td>85 (Sep 15-Nov 19)</td>
<td>3,981</td>
<td>253</td>
<td>14</td>
<td>960</td>
<td>667</td>
<td>100</td>
<td>100</td>
<td>65</td>
<td>93</td>
</tr>
<tr>
<td>kumulus</td>
<td>Multi-Repo</td>
<td>Java, JavaScript</td>
<td>94 (May 15-Nov 19)</td>
<td>2,487</td>
<td>82</td>
<td>19</td>
<td>2,083</td>
<td>274</td>
<td>100</td>
<td>100</td>
<td>45</td>
<td>22</td>
</tr>
<tr>
<td>jhipster</td>
<td>Multi-Repo</td>
<td>Java, JavaScript</td>
<td>71 (Nov 15-Nov 19)</td>
<td>10,931</td>
<td>767</td>
<td>11</td>
<td>7,166</td>
<td>3,627</td>
<td>100</td>
<td>100</td>
<td>64</td>
<td>59</td>
</tr>
<tr>
<td>netwont</td>
<td>Multi-Repo</td>
<td>Java, JavaScript</td>
<td>36 (Sep 16-Nov 19)</td>
<td>5,369</td>
<td>110</td>
<td>12</td>
<td>1,027</td>
<td>2,605</td>
<td>100</td>
<td>100</td>
<td>74</td>
<td>36</td>
</tr>
<tr>
<td>flume</td>
<td>Mono-Repo (Jira)</td>
<td>Java</td>
<td>115 (Jun 10-Jan 20)</td>
<td>28,253</td>
<td>1,199</td>
<td>16</td>
<td>3,483</td>
<td>3,638</td>
<td>51</td>
<td>57</td>
<td>29</td>
<td>86</td>
</tr>
<tr>
<td>stormd</td>
<td>Mono-Repo (Jira)</td>
<td>Java</td>
<td>107 (Nov 10-Nov 19)</td>
<td>6,933</td>
<td>149</td>
<td>21</td>
<td>17,469</td>
<td>1,490</td>
<td>54</td>
<td>80</td>
<td>47</td>
<td>62</td>
</tr>
<tr>
<td>falcon</td>
<td>Mono-Repo (Jira)</td>
<td>Java</td>
<td>67 (Nov 11-Mar 19)</td>
<td>2,556</td>
<td>190</td>
<td>19</td>
<td>7,962</td>
<td>2,760</td>
<td>56</td>
<td>58</td>
<td>47</td>
<td>63</td>
</tr>
<tr>
<td>reactor</td>
<td>Mono-Repo (Jira)</td>
<td>Java</td>
<td>154 (Mar 15-Jan 20)</td>
<td>5,446</td>
<td>1,327</td>
<td>21</td>
<td>9,609</td>
<td>5,353</td>
<td>71</td>
<td>65</td>
<td>46</td>
<td>67</td>
</tr>
<tr>
<td>spigot</td>
<td>Mono-Repo (Jira)</td>
<td>Java</td>
<td>155 (May 06-Jan 21)</td>
<td>7,125</td>
<td>831</td>
<td>27</td>
<td>10,696</td>
<td>2,649</td>
<td>54</td>
<td>73</td>
<td>57</td>
<td>68</td>
</tr>
</tbody>
</table>

as multiple individuals in the data set, hence the need for
deduplication (Fig. 2 (3)). We used the Dedupe library [17],
which uses machine learning to perform deduplication. It has
an accuracy up to 95% [18]. More details of the training
process are available in our dataset [15].

D. Extraction of Commit-To-Issue Relations

Github commit messages often refer to issues to identify the purpose of the changes. We manually inspected each project to identify what patterns developers tend to apply for referring to issue-id and applied that pattern when parsing the commit messages (Fig. 2 (4)). The following are two sample commits from our data referring to issue-id: FLUME-3311 and kumuluz#115 respectively.

- “FLUME-3311 Update User Guide In HDFS Sink”
- “Merge pull request #115 from Jamsek-m/master”

We applied the same mechanism to pull requests. Whenever a pull request identifies a Github issue, we merge the pull request and the Github issue into one dataset issue and link all commits that are part of the pull request to it. This is particularly helpful when commit messages lack a reference to an issue.

For the Jira-backed projects, we regularly find pull requests and commits without a reference to a Github issue but instead a reference to a Jira issue. Whenever a pull request identifies a Jira issue, we subsequently merged them into a single issue in our dataset (we never encountered multiple pull requests referencing the same issue) and link all involvements from the pull request and Jira issue to that merged issue. We proceed likewise whenever we find a Github issue referencing (in the comments or title) a Jira issue.

E. Mapping Subsystems

For those projects that utilize Apache Jira for issue management, aside from mapping Jira to Github users, we also need to map a Jira project’s components to the corresponding folders on Github (Fig. 2 (5)). We manually mapped each component of the five Jira projects to its corresponding Github folder or multiple folders where necessary. We link all those folders, which do not map to a component to the main system (we store main system as a subsystem in our data model).

As briefly mentioned above, the mapping for Github multi-repo projects is straightforward: each repository in the project becomes a subsystem. Furthermore, we store the list of folders that make up a subsystem. This allows us to relate all artifacts to a unique subsystem.

F. Linking Issues to Subsystems

Linking issues to subsystems (Fig. 2 (6)) works similar to relating artifacts to subsystems. Jira issues exhibit a Component property that identifies all affected subsystems (potentially multiple). Github multi-repo projects provide separate issue lists for each repository, thus we link those to subsystems unambiguously.

G. Dataset Overview

Table I provides an overview of the 10 chosen open source projects. The columns Type and Programming Language(s) report the repository structure and languages of the project, respectively. The column Time Period describes the overall time in months in which we extracted commits and issues. All subsequent rows report values from this time period. The Commits column reports how many commits are made in total in a project. Recall that in multi-repo projects, commits per subsystem belong to one repository, thus the reported numbers are the sum of commits over all repositories in such a project. In case of mono-repo projects, all commits come from a single, main repository. The Devs column provides the total number of developers in each project. We consider any person a developer who is Contributing or Informative in at least one subsystems. Subsys, Arts, and Issues columns report the number of subsystems, total source code artifacts, and issues in a project, respectively.

The Arts linked to Subsys column shows the percentage of artifacts belonging to subsystems. In a Mono-Repo system each folder has its link to a subsystem or to the main system. So in this case, the column shows the percentage of artifact which are linked to subsystems and not to the main system. Each repository (subsystem) of a Multi-Repo system has its own Issues, thus, the Issues Linked To Subsys column shows their 100% link to subsystem. In a Mono-Repo system, we also include issues from Jira, where they have specified mapping to a subsystem. As mentioned in the data preprocessing section, commit(s) are made against an issue and issue(s) are linked against commits. The Columns Issues Linked To Commit(s) and Commit(s) Linked To Issues specify the percentage of such linkage, respectively. Note that as the linkage is not always present e.g., commit messages not always refer to issues to identify the purpose of the changes. Therefore, the linkage percentage ranges between 22 to 93.
IV. A Technique for Measuring Community-SDT Alignment and Evolution

A. Developer Interaction Network Extraction

RQ1 requires building a developer interaction network. We built such a network from developers’ involvement in issues such as contributing code or commenting. We sliced the project lifetime into time windows of 4 months, as done in close-related works [10], [12]. We presume that such a time window is sufficiently long to allow developers to repeatedly interact to produce a reliable interaction network while being short enough to capture changes of the interaction network as developers join and leave.

Developer Interaction Network Model We model the developer interaction network (Fig. 2 (7)) as an undirected graph with developers as nodes \((d \in V)\) and edges \((e(d_i, d_h) \in E)\) between those developers that interact. We add an edge between two developers when the two developers are involved in the same issue, for example, one developer adding a comment to an issue, the other developer committing an artifact update linked to the same issue. The edge’s issue involvement intensity measures in how many issues the two developers are active in, weighted by their involvement type.

Developer Involvement Recall that the developer involvement types – Contributing and Informative – carry a weight of 3 and 2, respectively. We consider only the highest weighted involvement per developer and issue \(inv_{max}(d, i)\). The intensity for a single issue \(i_h\) and pair of developers \(d_i\) and \(d_j\) (i.e., the per-issue score) is then the lowest of the two scores \(\min\{inv_{max}(d_i, i_h), inv_{max}(d_j, i_h)\}\). For example, the per-issue score between a contributing developer and an informative one is 2. The overall intensity for a pair of two developers across all their common issues (i.e., the issue involvement intensity edge property) is then the sum of per-issue scores \(\sum_{i \in k} \min\{inv_{max}(d_i, i), inv_{max}(d_j, i)\}\). The issue involvement intensity serves as an edge weight.

Community Detection The subsequent step after forming the developer interaction network is to discover communities in it. The communities are the natural divisions of network nodes into densely connected subgroups [19]. The developer interaction graph serves as the input to the step of community detection (Fig. 2 (8)). We first remove all developers who have no edges to any other developer. We consider those as not part of any community and retaining them in the graph would negatively influence the community detection results.

For the purpose of community detection, we use the order statistics local optimization method (OSLOM) [20]. This method is also used in close-related work [12]. OSLOM finds statistically significant communities and has distinct features such as to handle weighted graphs, to form overlapping communities, and to distinguish communities from pseudo-communities.

Community Verification and Validation The OSLOM algorithm identifies the best distribution of graph nodes as communities after calculating the statistical significance (the probability of finding the cluster in a random null model, i.e. in a class of graphs without community structure) of each community. It uses the significance as a fitness measure to evaluate and include a community candidate in the resulting set of communities. We additionally checked the resulting quality of the community structure using conductance as follows.

Quantitative Community Quality Metric: We apply conductance [21], which measures the probability of having an edge leaving the community. A community with all edges connecting only member nodes has conductance 0 (an isolated community) while a community with no edges amongst its members has conductance of 1 (arguably not really a community at all).

Qualitative Detected Communities Inspection: In order to validate that the detected communities accurately reflect real-world developer collaboration, two of the co-authors manually inspected OSLOM’s detected communities of one of our projects, i.e., kumuluz. The project has 82 developers and 54 months of observed developers interactions. We inspected the detected communities for 5 time windows (20 months). We verified each detected community of every window by matching the strength of its collaborative relations compared to the other undetected communities. We also observed the correctness of developers division into communities based on edge weight.

The resulting communities consists of a subset of developers \((c_k \subset V)\) where developers potentially are members of more than one community (overlapping communities). This is an important property as in developer interaction networks, the most active developers are often connected to two or more communities [22]. In the absence of overlapping communities, a developer will be placed into a single community which will skew the subsystem developer overlap calculations.

Observed Time Windows We apply the community detection algorithm for each time window separately. We skip time windows with less than ten developers as we do not expect to find meaningful communities among such a small number of developers. Manual inspection of sample windows with less than 10 developers confirmed these expectations. For all these windows the best community structures found by OSLOM exhibited high conductance values (> 0.8).

B. SDT Extraction

Answering RQ2 requires identifying for each subsystem \((s_y \in S)\) who are the active developers: the SDT \((SDT(s) \subset V)\) (Fig. 2 (9)). To this end, we simply consider a developer as active if one has at least two Contributing or five Informative involvements in that particular subsystem within the defined time window. This ensures we select only developers that have a longer-running interest in a subsystem and not just include anyone with minimal involvement. We continued with these values i.e., two and five after observing the trend of the number of Contributing and Informative involvements in the chosen projects.

In contrast to the interaction network community construction, a developer becomes member of an SDT purely based on one’s contributions to source code and/or involvement in issues
regardless of interactions with other developers. For example, two developers changing the same artifact within the same time window that otherwise are not involved in any issue will end up in the same SDT.

Similar to developer overlapping communities, multiple SDTs may contain the same developer. Especially key developers are often involved in multiple subsystems. At the same time, subsystems may be too small to give raise to a dedicated SDT that focuses only on that single subsystem. Ultimately, for each developer we define the community membership \( cm(d) \in [1, k] \) and SDT membership \( SDTM(d) \forall S \).

C. Overlap Calculation

The second part for answering RQ2 is a metric for measuring how well a SDT matches a developer community (Fig. 2 (10)). In a project where the community structure represents subsystems, we would expect that members of the same SDT are also members of the same community (low membership heterogeneity). On the other side of the spectrum, we would find a SDT where every member belongs to a different community (high membership heterogeneity). We measure each SDT’s membership heterogeneity using the normalized Shannon entropy [23] \( mh(SDT(s)) = -\sum_{d}(p_kLn(p_k))/Lnk \) where \( p_k \) is the number of developers in \( SDT(s) \) being member of community \( c_k \) and \( Ln \) is the natural logarithm. The normalized Shannon entropy provides a result in the interval \([0,1]\): 1 when \( p_k \) is the identical for every \( k \) and 0 when \( p_k = 0 \) for all but one \( k \). In other words, \( mh(SDT(s)) = 1 \) when all SDT members are exactly equally distributed across all communities, and \( mh(SDT(s)) = 0 \) when all SDT members are member of the same community. In our fictive example in Fig. 1 \( mh(SDT) \) for Subsystem A is 0 and \( mh(SDT) \) for Subsystem B is 1. Ideally, we observe minimal/low membership heterogeneity for all subsystems.

Having introduced the general concept of membership heterogeneity, we need to outline an adjustment to \( p_k \). Without any adjustment each developer has equal impact. However, this does not properly reflect the different developer types in a typical open source project. There is a small number of key developers that get involved in issues and source code changes across the subsystems, subsequently being well connected and informed. Giving them equal weight as a developer who is focused only on one single subsystem would skew the measure. We thus redefine \( p_k \) as the weighted sum of developers in \( SDT(s) \) being member of community \( c_k \) where the weight describes the developer’s focus \( f(d) \in [0,1] \) across the subsystems (no other adjustment to \( mh(SDT(s)) \) needs to be made). The focus on a single subsystem is maximal when the developer is active in a single subsystem, and minimal when involved in every subsystem. To this end, developer focus is calculated as 1 minus the normalized Shannon entropy of the developer’s involvement in each subsystem (see Subsection IV-B); hence \( f(d) = 1 + \sum_y(p_yLn(p_y)/Ln(y)) \) with \( p_y = \sum_{d \in y} inv_{max}(d, i) \).

D. Membership Evolution

Regarding RQ3, we expect community structure as measured by conductance, and subsystem alignment as measured by SDT membership heterogeneity to change over time. Focusing on community structure, we want to be able to interpret community evolution not only through changes in conductance but also by looking at the external stability of developers (joining and leaving developers) and internal stability (shifting of developers from one community to another in two consecutive time windows \( w_t \) and \( w_{t+1} \) (Fig. 2 (11)). Measuring external stability is simply a matter of tracking which developers appear as nodes in the interaction network graph in one time window but not in the next and vice versa. As the detected communities have no meaningful, intrinsic identity (we just assign them a number as identifier) internal stability requires tracking which developers are in the same community in one time window and remain in the same community (irrespective of the community identifier) in the next window. To this end, we counted how many pairs of developers were in the same community in \( w_t \) and are again in the same community in \( w_{t+1} \) (sameC) or were in different communities in \( w_t \) and are in different communities again in \( w_{t+1} \) (diffC). The internal stability metric is then \( iStab = (\#sameC + \#diffC)/totalPairs \) and yields 1 for full stable communities and 0 when all communities become completely reshuffled.

Similar to developer community membership evolution, we are interested to know whether the SDTs that overlap with the same community in one time window will overlap with the same community in the subsequent time window (Fig. 2 (11)). We apply the same calculation as for internal stability above but instead of comparing a pair of developers being in the same or different community again, we measure in \( sdtStab \) whether pairs of SDTs tend to overlap with the same community in the subsequent time window again or whether they overlap with different communities.

V. Results

In this section, we present the detailed results for a single example project flume only due to page limitation. We thus provide aggregated numbers across all ten projects and refer for per-project details to our supporting online material (SOM) [24]. The SOM includes a database dump of the underlying dataset, the source code for extracting the raw data, preprocessing, and metric calculation, as well as detailed figures for each project.

A. Answering RQ1

For answering RQ1: To what extent can we identify well defined development communities [. . .] we analyse for each project how many communities we find for each time window and their conductance. Fig. 3 displays the count, size, and average conductance of communities found for the example flume project across the observed time windows (here twenty six windows (window-20) has less than 10 active developers therefore not included). The number of communities ranges between one and eight communities. Note that as communities
are latent – they do not have an explicit identifier but can be observed – community 4 in window 11 might be most similar to community 5 in window 12 in terms of overlapping members. The size of a community, i.e., number of developers in it, varies as well. As shown in Fig. 3, often communities exceed typical team sizes of ten members (as also previously observed [12], [25]) as communities also include rarely and non-contributing members. These rarely-contributing members are significant for estimating software quality (e.g., defect prediction) [25].

We notice that the mean conductance for each project is below 0.4 except networknt and falcon, with no time window of any project exceeding a conductance of 0.6, a range similar to previous observations for such a developer count [12].

Fig. 1 reports the range of community count $k$. For all projects the number of community count ranges between one and eight.

Observation 1: There are far fewer communities that emerge from the developer interaction network than subsystems (e.g., flume has 16 subsystems).

Observation 2: Overall, communities tend to have medium to low conductance, thus the observed developer interaction networks exhibit a clear community structure.

B. Answering RQ2

For answering RQ2: Do the developers active in the same subsystem emerge in the same development communities? we calculate SDT membership heterogeneity for each subsystem in each observed project for all time windows with at least 10 active project developers and at least two communities (heterogeneity between a single community and SDT is 0).

Fig. 6 compares the range of average SDT membership heterogeneity across all intervals per project, i.e., we averaged the SDT membership heterogeneity metric over all subsystems where that metric was calculated. The box plots in Fig. 6 then describe for each project the range of that heterogeneity average when calculated for each time window.

We notice a wide range of heterogeneity behavior across the projects. While nameko, networknt, flume, and falcon display low average heterogeneity, hipster, and stanbol yield rather high average heterogeneity. The remaining projects yield a medium range of mean average heterogeneity, yet have the occasional time window where average heterogeneity reaches high values, i.e., up to 0.7.

With significant spread of heterogeneity values across subsystems (in flume, for example, between 0.0 and 0.71), we investigate whether high heterogeneity might be correlated with large SDTs (i.e., the more members in a team, the more likely they come from different communities). The data show a weak correlation between SDT size and heterogeneity with a Pearson’s correlation coefficient value of 0.46. We observe this phenomenon also in the remaining projects.
We subsequently investigate whether SDTs form their own, small communities that OSLOM did not detect. Such communities may form when developers interact primarily with other developers working on the same subsystem and not so much with developers of other subsystems. To this end, we treat each SDT as the members of a virtual community and calculate its conductance.

Fig. 7 displays SDT size for each subsystem in the example *flume* project. For easier interpretation, conductance values are provided in Fig. 8 next to it. We observe that SDTs exhibit high conductance values (close to 1 and darker in color), thus they are tightly integrated in the developer interaction network and hence do not represent well defined communities. Recall that, a community with no edges amongst its members has conductance of 1 (arguably not really a community at all).

**Observation 3**: Projects are diverse with respect to how well SDTs typically overlap with a single community. All projects exhibit intervals in which SDTs experience high heterogeneity.

**Observation 4**: SDTs do not form subcommunities in the developer interaction network but rather maintain considerable interaction ties with developers not involved in the particular subsystem.

C. **Answering RQ3**

For answering [RQ3]: *How stable are the detected communities across time compared to the SDTs?* we need to measure internal stability of the communities between two subsequent intervals to understand how much the developer interaction network structure itself changes. We then compare the internal stability with the SDT stability.

Fig. 9 reports the evolution of the example project *flume* developer community membership as measured by external and internal stability (interval 1 reports the change from interval t to t+1). We notice a heavily fluctuating number of unchanged developers (between 0 to 27) while an even higher number of developers keep joining (between 5 and 58) and leaving (between 9 and 51). This comparatively large set of fluctuating developers influences the community structure, hence few stable key members substantially shift between newly forming communities. This is reflected by the internal stability metric drifting between 0.27 and 0.65. The SDT stability remains above that between 0.65 and 0.91.

For every project we select the time windows with at least 10 active developers. We then determine the internal stability of developers across communities (Fig. 10). We also determine the stability of SDT overlap with communities (Fig. 11): i.e., whether two SDTs mostly overlapping with the same community in time window t do so also in time window t +1.

**Observation 5**: In general, we find that internal stability is not that strong, hence implying that communities change considerably across time. In contrast, we find considerably stronger SDT stability. This indicates that developers remain in the same SDT but switch among communities (i.e., pre-
VI. DISCUSSION AND IMPLICATIONS

The findings of our study are that well defined communities emerge for time windows with \( \geq 10 \) developers. We observe that the alignment between these communities and SDTs covers the complete spectrum from high to low overlap with no discernible influence of the SDT size. Hence, we draw the conclusion that there is no correlation between communities and SDTs: working on a subsystem does not cause interaction links to occur predominantly among SDT members.

We make the careful hypothesis that the low overlap is to mitigate the high developer fluctuation. Interaction links among members from multiple SDTs enable to better cope with situations when developers leave. Then the burden to take over or on-board new members is not placed only with members of that SDT. Cross-SDT interaction may thus be a mechanism to remain robust in the presence of high fluctuation. The low community stability can then be explained as the remaining developers rearrange (at least partially) their interactions and coordination around newly joining members (and the effect of leaving members).

The significant interactions across SDT boundaries also raise the question whether a microservices-centric approach may be suitable in the context of open source software development. The perceived advantages of microservices, among others, are a reduction of coordination needs across microservice-centric teams (i.e., SDTs) [13]. Assuming our hypothesis outlined above is correct, the open source community might benefit, even need, such cross-SDT interaction and hence might not be able to benefit fully from a microservice-centric approach. We are not expecting such high overlap in industrial settings – a subject of future studies.

Researchers and Practitioners Implications Our work provides a mechanism to measure development communities and their overlap with SDTs. We provide evidence of heterogeneity of development communities and where evolving sub-communities do not overlap with the subsystems compositions. This approach is potentially useful for practitioners, e.g., lead developers in open source projects, to assess the interaction structure among developers and potentially identify subsystems where teams are less well connected. Similarly, practitioners in industrial setting may find the approach useful to identify subsystems where teams are highly interacting, thus perhaps identifying inadvertently strongly coupled subsystems.

A. Threats To Validity

Construct Validity We generated the developer interaction network from issues and commits linked to those issues to identify developers that coordinate their work. We did not contact developers to verify the detected community structure as we do not expect them to accurately recall differences in community structure for fine-granular time windows of 4 months for several years into the past.

Besides issue trackers and commits, we did not consider other possible developer communication channels, i.e., mailing lists, IRC, and conference calls, which could impact the links created among developers. We investigated that among our analysed projects only Jira-backed projects use mailing lists. Panichella et al. [11] investigated how developers collaboration links vary when data is gathered from different sources. They found, on the one hand, that communication links obtained from mailing lists have high overlap with links obtained from issue trackers and, on the other hand, that emails as the primary communication channel is increasingly replaced by chat and issue trackers. Other communication channels such as IRC and conference call are also used in practice. Four of the analysed projects use a chat platform (i.e., Gitter\(^3\) where conversations often happen in private threads which cannot be mined). However, research [11] has shown that mining links from chat is less reliable as 1) this tends to produce too many links, and 2) conversations are less easily associated with issues. We therefore believe that we miss only negligible information by leaving out mailing list and chats. We, however, like to point out that co-located developer interactions are not considered and hence restrict the applicability of our approach to projects with completely distributed developers.

Internal Validity In order to address the internal validity threat, we analyse data from multiple open source systems rather than conducting controlled experiments. The analysis focused on commits and issues in general and was not specifically tailored to Java projects. We sampled the chosen projects based on different subsystem structure mechanisms: mono-repo and multi-repo based to avoid a single mechanism influencing the SDT stability or evolution thereof.

---

\(^3\text{https://gitter.im/}\)
Some of the chosen projects show a low percentage of linkage between commits and issues, thus a threat to the validity of the study. However, we observed similar community quality metric results of 10 additional projects with higher linkage values (the results are a part of the SOM [24]). These projects are not included in our study as they do not completely fulfill other inclusion criteria mentioned in the Section III.

We cannot be absolutely certain that the mapping between Jira issue components and Github folders is absolutely correct. A few Github folders for the five Jira-backed projects that could not be mapped to a component with high confidence were excluded to avoid skewing the results.

External Validity Generalizability of our observations is limited to open source projects that make significant use of issues (incl. pull requests) as their primary coordination and communication mechanism. We do not expect our insights to apply to commercial projects where developer teams are top-down defined rather bottom-up emerging. The developers in commercial projects are typically (at least partially) collocated. Even when a significant amount of communication happens online, a significant amount of coordination is expected to occur off-line, thus resulting in a less accurate dataset with respect to communication.

VII. RELATED WORK

Several researchers have analyzed developer interaction data from various sources, such as version control systems, mailing lists, or issue trackers, to investigate how emerging development teams are formed in open source projects.

Joblin et al. [12] provide an approach to identify the community structure of a software project. They capture a view on developer coordination, based on commit information and source-code structure. We applied the same algorithm for identifying communities, however, we applied developer interactions based on their involvement in issues along with commit information. Moreover, they capture a macro-level view (once for the overall project) of coordination compared to our approach which focuses on a more fine granular level (developer interaction network alignment with SDTs at separate snapshots in time).

Researchers also examined the effect of interaction between developers on software quality. Tamburri et al. [26] [27] explored the relation between community smells and code smells in open source environments. They base the detection of community smells on the micro-granular structural differences (i.e., non/existence of edges between individual developers) between the collaboration network generated from commits and the network derived from developer interactions in issues or mailing list. They also propose YOSHI [28], a tool to monitor key community traits in open-source projects. In contrast, our approach observes the community level but not patterns of edges around individual developers.

Bird et al. [25] examined the effect of different ownership measures on release failures in an industrial setting. They found that social network metrics are useful predictors.

Leibzon [29] studied the organization of software development teams and project communities at Github. Nzeko’o et al. [30] made a social network analysis and comparison of developers’ and users’ mailing lists of four open source software projects. Similarly, Bird et al. [3] analysed the latent social structure by forming a social network from the project’s mailing list. They showed results that sub-communities arise within a project as the project evolves.

Several efforts aim to exploit the information embedded in the social structure. Canfora et al. [31] mined explicitly mentioned cross-system bug resolutions and correlated these activities with social attributes of developers who participated in discussions, e.g., developer mailing lists and commits in source code. Their study showed that cross-system bug fixing mainly involves developers who engage the most in mailing list interaction and developers who are among the top committers. Mockus et al. [32] used email archives of source code change history and issue reports to quantify aspects of developer participation, core team size, code ownership, and productivity for open source software projects.

Panichella et al. [33] studied how emerging teams evolve over time and that these teams tend to work on more structurally and semantically related set of files. Whether these files belong to or represent a particular subsystem was not part of the study. Hong et al. [34] also conducted a study to understand developer social network and its evolution. They observed that developers and their relationships change continually. Avelino et al. [35] studied how authorship-related measures evolve in open-source communities. While they focused only on one huge system, i.e., the Linux kernel, we in contrast focused on several comparatively smaller open source systems.

To the best of our knowledge, no approach studied how emerging communities align with subsystems, nor how stable SDTs are over time compared to developer communities.

VIII. CONCLUSION AND FUTURE WORK

In this paper, we investigate the emergence of latent developer interaction communities and how they align with subsystem developer teams (SDT). We observed that developer community membership is less stable than subsystem developer teams. We noticed hardly any correlation between detected communities and SDTs. One cause behind the observed low overlap between communities and SDTs could be the need to remain robust against high developer fluctuation in an open source development environment.

As a future work, we are interested in comparing in more detail those projects that experience different levels of developer fluctuation, respectively what other factors might bring about the lack of correlation between communities and SDTs.

ACKNOWLEDGMENT

This work was supported by the Austrian Science Fund (FWF): P31989 and P29415-NBL, and the LIT Secure & Correct Systems Lab funded by the state of Upper Austria.
REFERENCES


Bibliography


