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 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, customizable 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, for which the high number of projects, issue types, and instantiated states 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 assignee 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 <$Open by UserRole, Open by UserId, Assignee by UserRole, Assignee by UserId, Assignee to UserId$>. 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 by UserType equals In Development by UserType), values that are equal for a majority of all sequences (e.g., Open by UserType 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 regarded 69.3% of all constraints as useful. The ratio of useful 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%,
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-living 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

![Figure 5. Constraint appearance across time windows (RQ4).](image)

![Figure 6. Total number and number of window-exclusive constraints extracted for each time window in project P2 (RQ4).](image)
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 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. [9] 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 Foundation2 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 issues3 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 a 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

---

2https://issues.apache.org/jira
3https://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, CON- FIG, 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. C ONCLUSION

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


