Process Mining from Jira Issues at a Large Company

Bavo Coremans*†, Arjen L. Klomp*, Satrio A. Rukmono†, Jacob Krüger†, Dirk Fahland†, Michel R. V. Chaudron†
**Thermo Fisher Scientific*, Eindhoven, The Netherlands
{ bavo.coremans | arjen.klomp }@thermofisher.com
†*Eindhoven University of Technology*, Eindhoven, The Netherlands
{ s.a.rukmono | j.kruger | d.fahland | m.r.v.chaudron }@tue.nl

***Abstract*—Maintaining a software system is a continuous and complex process, typically following a workflow defined by the responsible organization. However, in practice, developers often deviate from the defined process due to personal preferences, varying customer requirements, or urgent deadlines. Such deviations may cause problems later on, or they may indicate potential process improvements. To deal with deviations and improve processes, it is first necessary to fully understand the processes actually employed by developers. For this purpose, process-mining techniques have been proposed that primarily build on version-control data. In this paper, we present a complementary process-mining technique that uses Jira issues to recover process activities not visible in version-control data, particularly focusing on developers' interactions with issues and each other. We conducted a case study with 74 repositories of 24 developer teams from one large international company to understand the technique's merits. Our technique revealed process differences across teams and depending on the types of Jira issues, providing novel insights for the company that helped to better understand the employed processes.**

***Index Terms*—Process Mining, Jira Issues, Software Development, Maintenance, Evolution**

I. Introduction

Developing and maintaining a software system is a socio-technical process that requires collaboration and interactions among developers and other stakeholders [9], [30]. While standard processes exist for software development, such as waterfall, agile, or platform-engineering processes [4], [9], [16], [20], [25], the actual process employed by a team of developers in an organization may deviate or evolve from these standards [15], [22], [27], [28], [31], [32]. For instance, the standard process may not be clearly communicated, or the developers may deviate from it to meet an urgent deadline. Such deviations can cause inconsistencies in the system's documentation, delays in follow-up activities, unfulfilled requirements, or miscommunication.

To tackle deviations and improve processes, it is important to identify and analyze the processes that are *actually* employed in an organization, in contrast to the processes *assumed* to be used. Researchers have proposed various techniques for mining and visualizing processes from different sources, primarily using version-control data [13], [19], [26], [29]. Most of such techniques focus on version-control data because it is automatically stored, is of high quality, and reliably represents what developers are doing. However, such techniques may miss activities outside of version-control systems, such as developers' interactions. Thus, a few works have also integrated other sources like issue-tracking systems [12], [14].

In this paper, we introduce a complementary technique for process mining from Jira issues that we developed for and have used at Thermo Fisher Scientific (TFS). Jira issues involve documentation that can provide further information on a team and its interactions (e.g., urgency of a feature request, responsible developers, issue status) [17]. We derived our technique based on a typical design-science workflow [8] and evaluated it via a multi-case study [38]. Our case study involves data from 24 developer teams working on 74 repositories for more than one year at TFS. The results show that our technique can be used to mine process activities outside of version-control systems, and it helped identify differences between the processes employed by teams as well as for different types of issues.

More specifically, we contribute the following:

- We describe a complementary technique for process mining that relies on Jira issues (Section II).
- We evaluate our technique by employing it in a multi-case study on Jira issues from 74 software repositories of 24 developer teams at TFS (Section III).
- We present the results of our case study (Section IV) and discuss their implications (Section V).
- We share a prototype of our technique in a persistent open-access repository [5].

Our technique complements existing research (Section VII), and others can build on it in the future. The results of our case study indicate that our technique can help practitioners inspect, analyze, and improve processes.

II. Mining Processes from Jira Issues

Next, we describe our process-mining technique, which we developed based on recommendations for design-science research by Dresch et al. [8] and for process mining by Aalst [34].

A. Context, Problem Definition, and Proposed Solution

We conducted this work as an industry-academia collaboration at TFS, a large international company that operates in the domains of laboratory equipment, biotechnology, and healthcare. With a global workforce of over 130,000 employees and generating more than $39 billion in revenue in 2021, TFS is listed in the S&P 100. We collaborated with a department responsible for the development and integration of software for electron microscopes, located at TFS's office in Eindhoven (The Netherlands). This collaborative partnership allowed us to iteratively design our technique in a real-world context, including tests of intermediate prototypes, gathering feedback on the results, and conducting a multi-case study to gain insights into the mined processes (cf. Section III).

During an ongoing collaboration, we encountered challenges related to understanding details about TFS's processes and the causes for deviations we observed across different teams. For example, it was unclear why certain Jira issues remained in a particular state or how specific customer requests, such as urgent bug fixes, were handled among developers. To address these challenges and improve our understanding of the processes at TFS, it was crucial to gain a comprehensive overview of the current processes and potential deviations from the intended ones. However, we also experienced that existing solutions for process mining (cf. Section VII) were less effective in eliciting the processes. We discovered that the most significant deviations and concerns for the developers arose not during the implementation phase represented in the version-control system but rather in their interactions with each other and with Jira issues. So, we defined our research objective as developing a technique that can recover and visualize processes from Jira issues used by the teams. By focusing on the interactions and communication patterns among team members through the lens of Jira issues, our technique is intended to provide a better understanding of the actual processes employed by the teams.

Jira[1] is a widely established issue tracker with integrated tooling, such as project management via Kanban boards [21], [23], [24]. Therefore, any technique capable of mining processes from Jira issues can assist practitioners and researchers in comprehending their development processes. Specifically, Jira issues can provide a reliable basis for mining information on developers' activities, for instance, regarding which issues are addressed when by whom. We conducted multiple iterations together with TFS to design, develop, and test our technique and visualizations. In the following, we describe these two artifacts.

B. Development: Mining Processes

Structure of Jira Issues. To design our technique, we first needed to understand the structure of Jira issues. For this purpose, we investigated the official documentation[1] and how Jira issues are used by the developers at TFS. At TFS, every task has a corresponding Jira issue with an identifying key for referencing. That key comprises a team prefix followed by a number, with each team having its own board in which their respective issues are listed. For its Scrum workflow, TFS builds on five default issue types defined in Jira:

Epics represent large tasks that involve multiple stories and are typically used for a large feature or an overarching goal a team is pursuing.

Stories represent a stand-alone task or a task that is part of an epic. Each story should represent a piece of work that does not need to be broken down further, even though individual teams may decide to do so.

Bugs represent a problem identified in the software.

Tasks represent work that needs to be done ad hoc; in contrast to stories, which are usually created for planned work.

Sub-tasks represent small tasks that are part of a larger one.

During our domain analysis, we found that the developers at TFS mostly used the types *epic*, *story*, *bug*, and *sub-task*. Furthermore, issues can be linked to each other, for instance, via parent-child relationships indicating that an issue on a lower level of the hierarchy is part of the issue above. Such links can also intersect the hierarchy and can be of one of 11 types: *blocks*, *relates*, *duplicates*, *references*, *clones*, *follows*, *treats*, *contains*, *causes*, *split from*, and *discovered by*. When verifying the data we collected at TFS, we used JQL queries, a web interface provided by Jira that accepts queries similar to SQL, to confirm that the task issue type is not used in the projects we investigated and that all issues of type sub-task have a parent issue and are not standalone tasks.

Each Jira issue is basically a structured document involving three parts. First, the most important fields of each Jira issue reside at the top, including a title, description, status, resolution, assignee, and reporter. Second, more specific information is provided, which involves a backlog classification, fix version, story points, sprint, priority, linked issues, commits, and merge requests referring to the issue. Finally, each issue reserves space for the developers to document additional information and provides an overview of the changes made to the issue fields.

Notation. We decided to use the Petri Net notation as a feasible representation for modeling processes, on the implementation as well as visualization level [33]. In this notation, circles represent places, rectangles denote transitions between places, and arrows connect these two elements by denoting the transitions (i.e., actions) that can be taken from one place to reach another. For our technique, a place represents the state[2] a Jira issue is in, for which the whole history is available through the log associated with an issue. Then, the goal of our process-mining technique is to recover the transitions between places from the Jira issues and their histories.

We use such a place-based representation to recover processes because it is not known in advance whether there are intertwined processes when reconstructing the process graph for a set of issues. The place-based reconstruction allows for the presence of multiple initial and final places. Moreover, developers can access pages within Jira that display issues in a specific place, such as all issues in the current sprint or only issues assigned to a particular person. A place-based process mining can provide insights into when issues lose employees' attention in Jira views. So, by using Petri Nets as a place-based representation, we can better represent the real world and obtain more in-depth insights. After multiple iterations, refinements, and test runs of our technique on data by TFS, we have implemented the following algorithms.

Creating Process Graphs. To mine processes from Jira issues, we use Algorithm 1. In principle, the algorithm takes a number of Jira issues as input (line 1) and creates an empty Petri net

[1] https://www.atlassian.com/software/jira

[2] A *state* includes, but is not to be confused with, an issue's *status*.

Algorithm 1 MineProcess

```
1: function MINEPROCESS(issues)
2:     $G \leftarrow$ empty Petri net
3:     for issue in issues do
4:         AddIssueToProcess($issue, G$)
5:     end for
6:     CorrectProcess($G$)
7:     return $G$
8: end function
```

graph (line 2). Then, the algorithm loops through all issues, adding each to the graph by calling Algorithm 2 (lines 3–5). Finally, the algorithm calls Algorithm 3 to correct some inherent mismatches between issues (line 6) before returning the created Petri Net (line 7).

To add an issue to the Petri Net graph, we utilize Algorithm 2 (line 1). The algorithm first identifies the state of the issue (line 2) and, if the state does not exist in the graph, adds it as a place in the Petri Net graph (lines 3–5). Then, the algorithm iterates through all actions in the issue's history from newest to oldest (line 6). It reverts each action on the current state s to obtain the previous state s' (line 7) and adds a corresponding place to the graph if it does not already exist (lines 8–10). Additionally, if there is no corresponding transition between s and s' at this point, the algorithm adds the action as a transition between the two places in the graph (lines 11–13). The algorithm then proceeds to the next action in the history and repeats the process (line 14). Finally, with the first action in the chronological order of the issue, the algorithm reaches the initial state of the issue and stops. Consequently, all issue states and transitions between them that have occurred in practice are added to the Petri Net graph, representing the process through which the specified issue progressed.

To determine the state of an issue (i.e., the places of the graph), we take into account various fields of a Jira issue, specifically the *issue type*, *resolution*, *status*, *assignee*, *fix version*, *linked commits*, and *project*. However, we do not use all of these values directly in the state representation. For certain fields, we consider whether they are filled in and how many values they contain. As one example, we count the number of versions filled in for the field *fix version*, while we directly use the values of the field *issue type*. If two states have identical representations, we consider them equal in our process graph, which may result in certain actions not transitioning an issue to a different place. Consequently, self-transitions may occur in the process when a team member updates an issue's description or changes non-observed fields. We derived this identification step during our testing at TFS, in which we also found that we had to exclude self-transitions.

Algorithm 2 adds each place across all issues only once. This results in the merging of places from different issues. For example, issues #1 and #2 may start at places P_1 and P_2 respectively, at some point converge into a common place P_c, but eventually diverge to P_1' and P_2'. The resulting graph allows issue #1 to traverse a path $P_1 \rightarrow P_c \rightarrow P_2'$, which

Algorithm 2 AddIssueToProcess

```
1: procedure ADDISSUETOPROCESS(issue, G)
2:     $s \leftarrow$ getCurrentState($issue$)
3:     if Place($s$) $\notin G$ then
4:         add Place($s$) to $G$
5:     end if
6:     for $action$ in $issue.history$ do
7:         $s' \leftarrow$ revert $action$ on $s$
8:         if Place($s'$) $\notin G$ then
9:             add Place($s'$) to $G$
10:        end if
11:        if Transition($s', s$) $\notin G$ then
12:            add Transition($s', s$) to $G$
13:        end if
14:        $s \leftarrow s'$
15:    end for
16: end procedure
```

does not occur in the actual data. To correct the graph, we split the places that represent the same state of different issues, adding one respective place for each issue and linking it to the appropriate transitions (Algorithm 3). As a result, we obtain a corrected graph that rectifies the incorrect merges and increases the precision of the graph, as described by Aalst [34].

To achieve this correction, Algorithm 3 iterates through all places in the obtained Petri Net graph (line 2). First, it computes the power set of outgoing transitions (*OTPowerSet*) and then removes the set of no transitions and the set of all transitions (lines 3–4), as it is not helpful to split off none or all transitions from a place. Next, the algorithm loops over each set of outgoing transitions in the outgoing transition power set (line 5) to identify the set of incoming transitions through which paths pass, ending up in one of the transitions included in the outgoing transitions set (lines 6–16). If all paths going through these incoming transitions are identical to those leaving through the outgoing transitions, the algorithm splits off the transition(s) from the place (lines 13–15). It is important to note that Algorithm 3 considers paths, not issues, passing through a place, as issues can traverse a place multiple times. The algorithm also stops splitting off transitions if a place has only one incoming or outgoing transition left, or when the set of outgoing transitions is as large as the total number of outgoing transitions (line 6). If transitions that were previously split off are no longer connected to the respective place (line 9), the algorithm skips to the next power set element. Finally, *splitPlace* is called to create a duplicate of a place, transfer the incoming and outgoing transitions of the old place to the new place, and handle duplication or self-transitions (line 14).

When incorporating more issues into a Petri Net graph generated by Algorithm 1 later on (i.e., extending or evolving it), we first reverse the changes made via Algorithm 3 by merging all places with identical state representations again. This step is necessary because Algorithm 2 assumes that all state representations of places in the graph G are unique. Once we have added the new issues, we need to reapply Algorithm 3

Algorithm 3 CorrectProcess

```
1: procedure CorrectProcess(G)
2:     for $place \in G.places$ do
3:         $OTPowerSet \leftarrow$ powerSet($place.outgoingTransitions$) sorted ascending by $size$
4:         remove set of all transitions and set of no transitions from $OTPowerSet$
5:         for $outTransitions \in OTPowerSet$ do
6:             if $|place.incomingTransitions| \leq 1$
                       or $|place.outgoingTransitions| \leq 1$
                       or $|outTransitions| = |place.outgoingTransitions|$ then
7:                 break
8:             end if
9:             if not all $outTransitions$ in $place.outgoingTransitions$ then
10:                continue
11:            end if
12:            $inTransitions \leftarrow \{t \mid t \in place.incomingTransitions, t.paths.outTransition \subseteq outTransitions\}$
13:            if $inTransitions.paths = outTransitions.paths$ then
14:                splitPlace($place, inTransitions, outTransitions$)
15:            end if
16:        end for
17:    end for
18: end procedure
```

to G to eliminate paths that are not observed in the data as described above.

C. Development: Visualizing Processes

Visually, a Petri Net graph consists of two shapes: Circles for places and rectangles for transitions. These shapes are connected by arrows to display the direction in which each path must be read. To make the graphs more suitable for visual analyses, we customized their styling. Please note that the coloring we used and display in this paper (e.g., Figure 3) is not necessarily color-blind or grayscale safe due to the multitude of colors. However, this can be customized in our technique, and the figures are still comprehensible.

Coloring. We color transitions to gain insight into who (e.g., a bot, engineer, product owner) performs what action. So, a rectangle's color in the displayed Petri Nets shows the role of the actors involved. If more than one actor role is involved in a transition, the rectangle colors are proportionally to the number of times each role performed the action. Originally, we label each place with the name of the respective issue's status, and places as well as transitions with how often they occur (i.e., differences in the numbers indicate paths that occur less often and are omitted). Due to confidentiality and for better readability, we omit the places' names in this paper. Instead, we derived (cf. Section III) a classification of the issue statuses and added respective background colors. Moreover, while initial places (i.e., the starting state of a Jira issue) have thicker black borders, final places have green stripes if issues end up "done" and red stripes if they end up "rejected."

Arrow Thickness. We represent the time that passes between places and transitions by varying the thickness of the connecting arrows. Specifically, the thickness of an arrow from a transition to a place represents the average duration between that action and the action following it. Conversely, the thickness of an arrow from a place to a transition represents the average duration between the action performed by the transition and the action performed before it. We add such temporal information only to transitions connecting two different places.

Layout. We use the `CoSE-Bilkent` layout algorithm [6] to generate the layout of the final Petri Net graph. However, manual rearrangements of places and transitions may be necessary to enhance the legibility of the graphs. Although we tested an improved and faster version of this layout algorithm with constraint support, it did not result in better layouts [2].

The final graph includes all places and actions that occur in the mined process, resulting in an overly detailed and large graph that contains sequences of places and actions visited only once. Such types of graphs are referred to as "spaghetti models" by Günther and van der Aalst [11], caused by less-structured processes or parts of processes. To solve this issue, they remove edges and nodes by clustering them together with neighbors. In our implementation, we can filter out places and actions not visited by more than a certain percentage of all Jira issues (i.e., five or ten percent in our case study), reducing the graph's size and facilitating the identification of the main issue workflows. This method improves the legibility of the graphs we present in Section IV [34].

III. Case Study

To assess the usefulness of our technique, we conducted a multi-case study [38], which we describe in this section.

A. Case Description

We conducted our multi-case study by mining processes of 24 teams responsible for integrating software components for

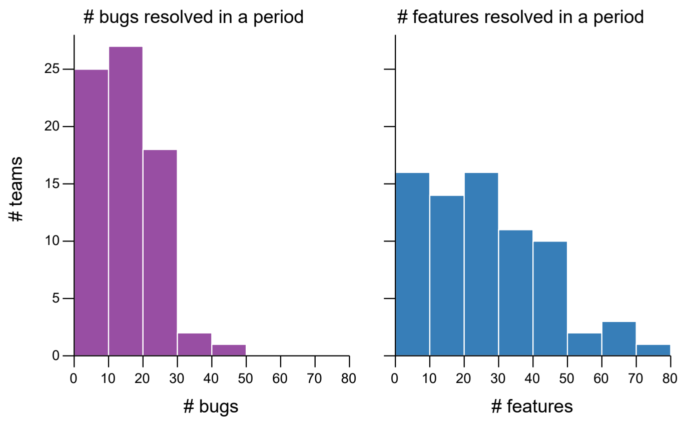

Fig. 1. Distributions of the number of bugs and features resolved per period.

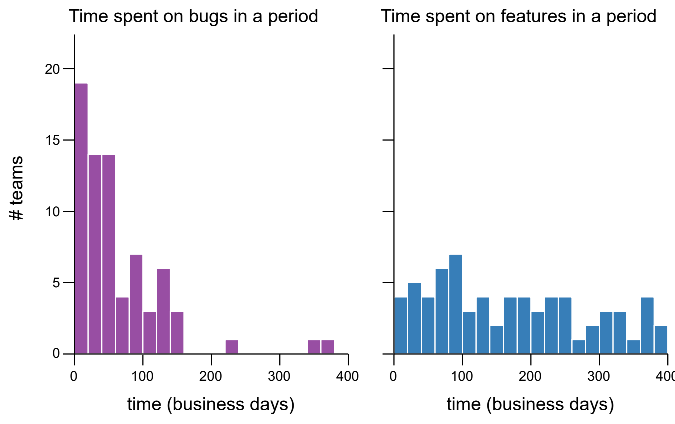

Fig. 2. Distribution of the time spent on bugs and features per period.

electron microscopes at TFS. For this purpose, we collected Jira issues from 74 repositories for five components of the electron-microscope software used by the teams. We excluded around 75 other repositories that had few commits (i.e., around 10 commits made by developers over 1.5 years), as they primarily stored rarely changing interfaces for legacy components. In contrast, the 74 repositories we selected involved multiple commits by developers per week or even per day. We analyzed data from March 8, 2021, to June 1, 2022, as the developers began to label their commits with relevant issue keys in early 2021, and a new release cycle started in March 2021. During this period, the developers created and resolved a total of 26,109 Jira issues for five software releases. Among these issues, we identified 14,739 non-duplicates, with 12,716 being created and 12,183 being resolved within this time span.

To simplify the mined process graphs, reduce complexity, and obtain more detailed insights, we separated bug and feature workflows due to the significant differences identified by inspecting first versions of the mined graphs. We only consider the workflow of the main bug or feature story in this paper (cf. Section IV) to further simplify the graphs, as including related issues and sub-tasks would introduce clutter. Additionally, we do not display all actions performed on bugs or features, as many issues are touched by more than 100 actions due to automated messages from the software integrated into Jira. We exclude actions that are not comparable across teams, such as differences in issue labeling names. Note that all of these adaptations are for the sake of readability and simplicity only. Our goal in this case study was to use the mined processes to shed light on the actual processes performed by the organizations' developers, helping them understand typical workflows and the causes for deviations.

B. Further Data Collection and Analysis

Besides employing our process-mining technique on the 14,739 non-duplicate Jira issues, we collected further data at TFS. We analyzed this data to provide an overview of our dataset and put our results into context. Overall, the following analysis underpins that our dataset of Jira issues represents a realistic and large-scale software system, which is why we argue that the teams represent appropriate cases for our evaluation.

Number of Bugs and Features. We recorded the number of bugs and features per team that have at least one linked commit during a specific time period. Each period here refers to the time the developers worked on one specific release, so the data builds on five different periods. In Figure 1, we show that almost all teams resolve 0 to 30 bugs within each period (i.e., between releases), with a few exceptions in the 30 to 50 bugs range. The teams commonly implement 0 to 50 features during each period, with the number of team-period pairs steadily declining as the number of features increases. Again, there are a few exceptions, specifically in the range of 50 to 80 features.

Time to Resolve Bugs and Features. We measured the time spent on resolving bugs and features in business/person days. Precisely, it is the sum of time that issues have the status "in progress," meaning that a developer is actively working on it. As we can see in Figure 2, multiple developers work in parallel, and a developer can have multiple tasks in progress concurrently. So, the total time in business days exceeds the number of days in the development period. More time is spent on implementing features than on bugs, with the time spent on features being distributed between 0 and 400 business days, and the time spent on bugs between 0 and 160 business days.

Bug and Feature Issue Structure. When discussing and investigating TFS's development processes with the team members, we found that the teams (intend to) follow standard processes. However, as we motivated, they were also aware that they deviated from these processes and that some parts of their workflow were unclear. So, we used our technique to understand how the following processes looked in practice.

The teams typically follow similar processes when handling **bug reports**. First, a bug is investigated by updating its description with additional information on how to reproduce it, sketching a solution, and identifying the software components it affects. After this analysis, the findings are presented to change and control boards (CCBs) to determine whether the bug should be fixed or rejected. Common reasons for rejection include that the bug is no longer relevant, that a fix is already

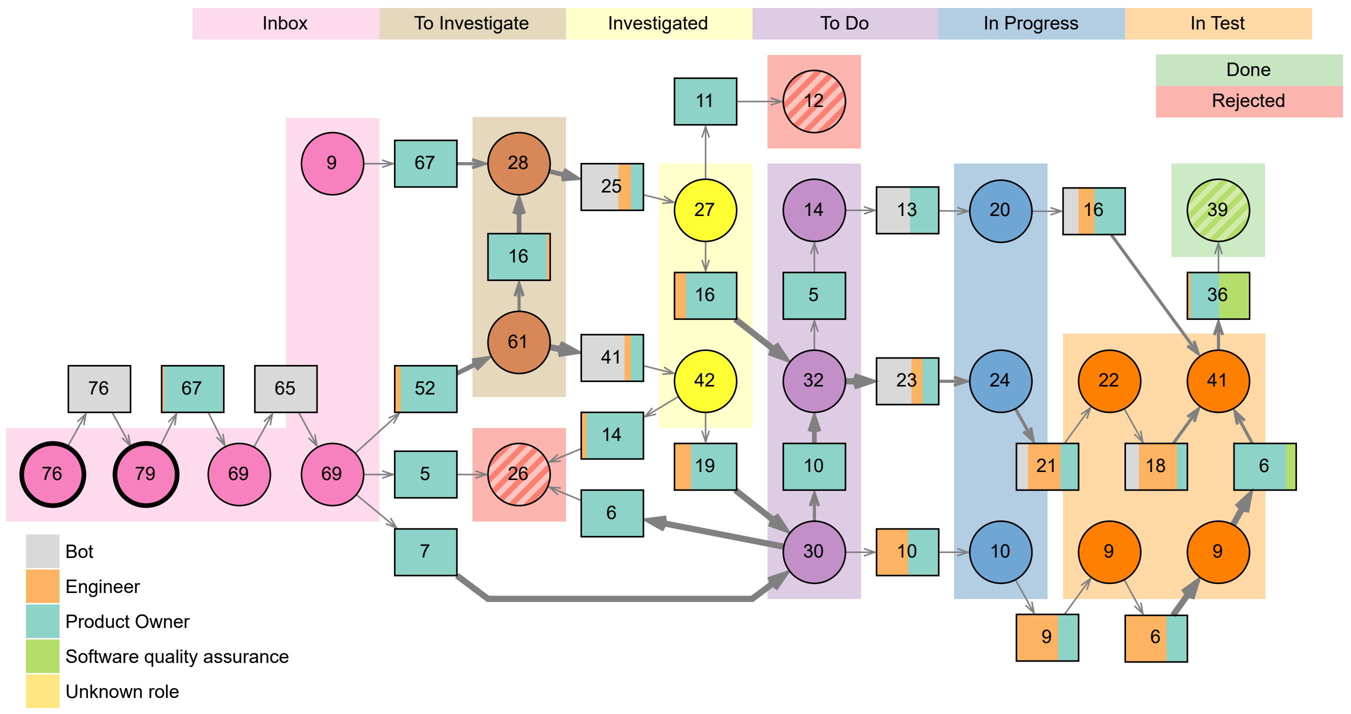

Fig. 3. The main bug-fixing process for team MOOSE. In this and subsequent figures, places and transitions are labeled with how many times they occur in the project for the duration. The numbers may not add up because some places are filtered out. We refer to Section II-C for this and more explanation about the features of the diagrams.

in progress, or that it actually represents expected behavior. If the bug is not rejected by a CCB, the team proceeds with fixing it and adds links to the commits and merge requests to the bug's Jira issue. The issue then moves to the status "in test" to verify whether it is resolved before being marked as "done."

Some teams document all actions on the original bug, while others create separate story issues for investigation and solution. For example, teams like ANTELOPE, MOOSE, and NEWT (names anonymized) create a story for the investigation but update the description of the original bug issue during that same investigation. MOOSE and NEWT also create one or more stories for a solution, depending on the size of the bug. In contrast, team ANTELOPE uses the bug issue itself to track the progress towards a solution, and in some cases, they fix the bug during the investigation phase if it requires minimal effort. Additionally, some teams choose to clone bug reports for every repository that requires a change (i.e., a cause for the duplicate Jira issues we found). During a qualitative inspection, we identified two types of issue links, namely "split to" and "clones" that are used to indicate relations between bugs and Jira issues linked to investigation or solution. We found that various other links are established for bugs to link, for example, bugs to related Jira issues that start with words like "solve" or "investigate," sub-tasks of bugs to their investigation or solution, and bug reports to epics (i.e., to group all bugs that are resolved in a version). For our case study, we used such links to identify duplicate issues and cleanse our dataset.

In the context of our analysis, **feature issues** refer to user stories or tasks that are independent of bug-related Jira issues. We did not include issues connected to bugs or linked to epics in our analysis of the feature-implementation processes. This allows us to focus specifically on issues directly related to adding new features or improving existing ones in a certain version—separating them from bug-related tasks and the broader epic-level initiatives (cf. Section II-B).

Issue States. By inspecting the mined process graphs, we manually abstracted specific issue states into more abstract categories. For this purpose, we investigated each individual place and clustered those representing similar concepts together, similar to a card-sorting methodology. We use these abstracted categories to provide an indication of the issues' states, while ensuring the confidentiality of the detailed data. Overall, we derived eight categories: *inbox*, *to investigate*, *investigated*, *to do*, *in progress*, *in test*, *done*, and *reject*. In the next section, we exemplify these categories and how the concrete activities they involve differ between the teams' processes.

IV. RESULTS

Next, we present our findings on the bug fixing and feature implementation processes we mined. We analyze the processes of two teams (MOOSE, NEWT) when fixing bugs and the feature implementation processes of two other teams (BISON, JAGUAR). In particular, we report insights and differences that were of interest to TFS for understanding their processes.

Processes for Fixing Bugs. In Figure 3, we illustrate the bug-fixing process of team MOOSE that we mined from our dataset. For this example, we display only places and transitions

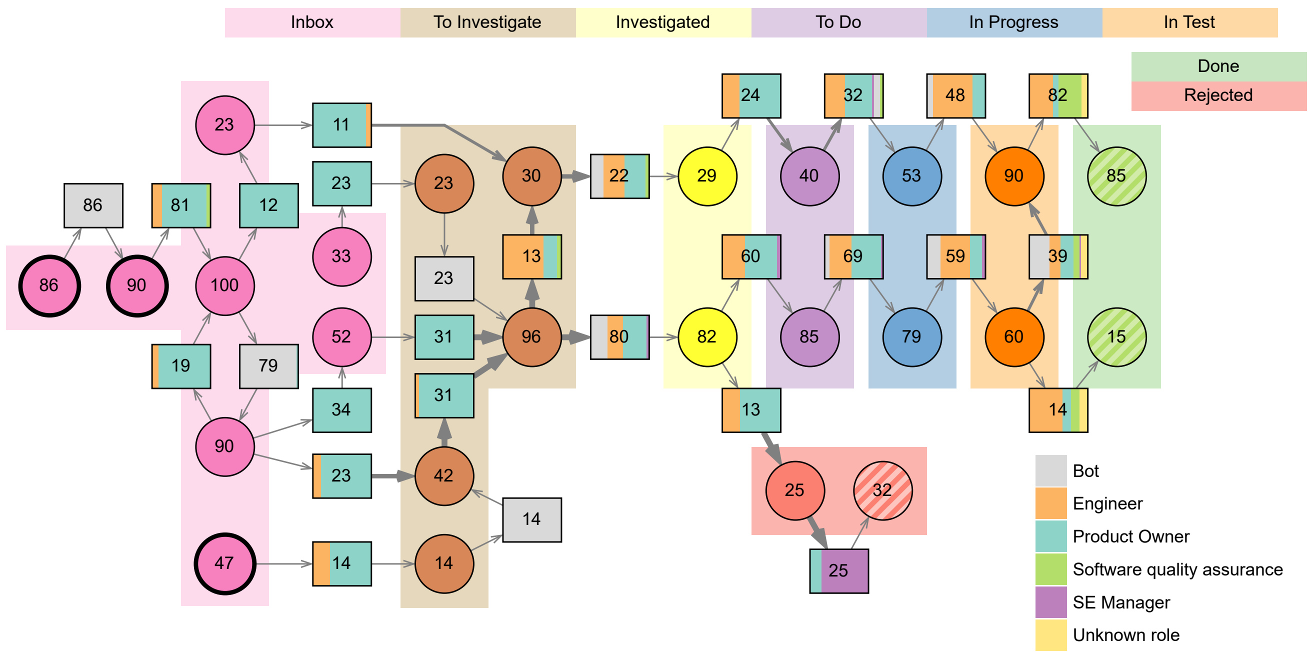

Fig. 4. The main bug-fixing process for team NEWT.

that represent at least five percent of the total number of the respective entities of that team and exclude self-transitions. The two initial places on the left-hand side correspond to the portals where bugs can be submitted, and such issues are pulled into the MOOSE repository by the product owner—indicated by the cyan transition. Essentially, the five places in the beginning all represent an *inbox* for bug issues in which any new bug report is collected. In most cases, issues move directly from the *inbox* to *to investigate*, while some issues go to *rejected* or *to do* (i.e., particularly severe bugs) places. Issues spend considerably more time in *to investigate* than in other places, with a mean time of 17 days, after which they always move to *investigated*.

After a developer investigated a bug, it may be *rejected* or marked as *to do*, a place in which some bug issues remain for a longer time, as depicted by the thicker arrows in Figure 3. The top two paths to the right of *to do* represent reproducible bugs that are assigned a fixed version and sometimes an assignee before moving out of this place. The path at the bottom represents intermittent bugs that occur infrequently and are not easily reproducible. In this part of the process, transitions are mostly initiated by engineers (orange). Bugs progress through *in progress* (i.e., the actual fixing) and then *in test*. For the latter place, they are assigned a responsible team member for testing, typically a software quality assurance team member (green), who initiates the transition from *in test* to *done* within MOOSE. Additionally, many transitions are automated by a bot (grey), indicating actions performed by software integrated with Jira. These automated transitions are triggered by rules set by the teams, such as moving an issue to *investigated* if a story linked to a commit starting with "Investigate" is marked as *done*. However, these automated transitions are not always executed correctly. For example, in team MOOSE, a team member may move a sub-task to *done*, but forget (and the automation fails) to move the respective story to *in progress*.

In Figure 4, we display the bug-fixing process of team NEWT. The process is often similar to team MOOSE, since both teams utilize standard bug workflow statuses for electron microscopes. However, we can see some differences. In addition to the initial places representing the bug submission portals, NEWT has an additional initial place representing bugs created within their Jira directly. By inspecting these issues qualitatively, we found that these bugs are reported by the team's test engineer and team members. Please note again that Figure 3 displays only paths that occur at least five percent of the places, so it does not imply that there are no issues created by the test engineer—although it is less common. Another difference is that NEWT adds a fix version while the bug is in the place *inbox*, earlier than team MOOSE. Furthermore, there is an absence of bugs that remain *in test* for an extended period. The mean time spent *in test* is three days, and the median is less than one day, which is much shorter than in the other team. Additionally, issues in team NEWT can move to *done* without ever having an assignee assigned, which is something that does not occur in team MOOSE. This seems to only happen for simpler bugs, as bugs without an assignee move to *done* faster. When examining the timing of the two paths from *to investigate* to *done* with an assignee, it does not make a difference at which point the assignee is added.

During the process of team MOOSE, issues spend most of their time in *to investigate* and *to do*. However, long waiting times are less concerning in *to do*, because teams cannot work on everything simultaneously. It was interesting to see such differences in the waiting times, and TFS considers this an important point to investigate in the future. In contrast,

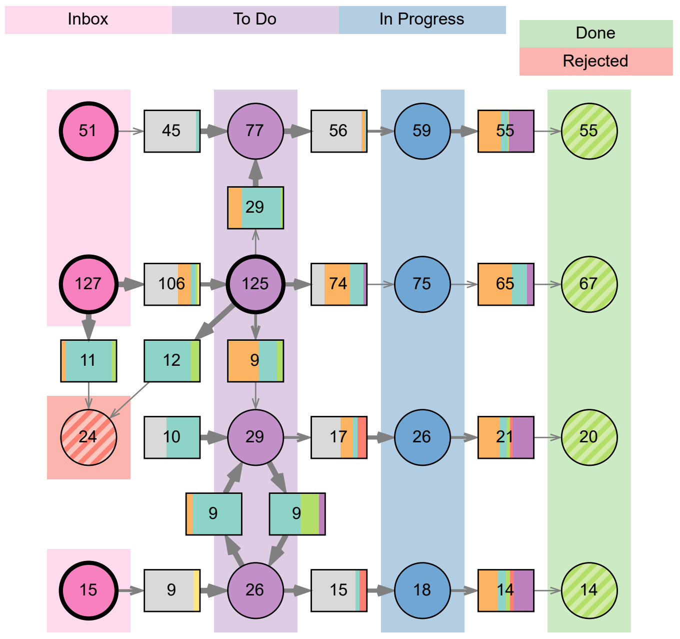

Fig. 5. Feature implementation process for team BISON. We omit the legend for brevity; the scheme is identical to the other process figures.

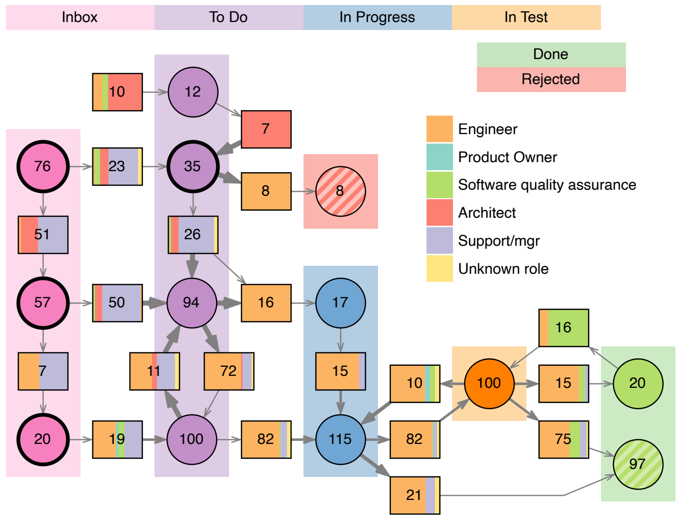

Fig. 6. Feature-implementation process for team JAGUAR.

team NEWT does not have long waiting times in *to do*, which suggests that once a bug is investigated and not rejected, it gets picked up almost immediately, resulting in a median time of less than one day spent in this place. This was an interesting insight for TFS, and we aim to understand whether this situation is caused by the complexity of the component, bug report, or any other factor to understand how to decrease waiting times.

Processes for Implementing Features. Compared to the bug-fixing processes, the feature-development process of team BISON consists of fewer places. We illustrate the mined process in Figure 5. Typically, issues start in one of three initial places with or without a fixed version and assignee and can move to *to do* with both fields unfilled, mainly for overdue administrative reasons. The mean time for issues being *in progress* without a fixed version or assignee is four seconds, compared to ten days for those with these values assigned. Unlike bugs, features developed by team BISON are not tested by someone else before being moved to *done*, so there is no status *in test* in the process. Verification sometimes occurs as a sub-task of a story, but it is not part of the typically employed process. In general, team BISON has a flexible workflow, with fields commonly left unfilled and no strict order in which actions take place. As with bugs, features spend a considerable time in places with a *to do* status.

In contrast, team JAGUAR employs a more determinate process, which we display in Figure 6. All issues that end up *done* have a fixed version and an assignee, and only features with a fixed version added move to *in progress*. If no assignee has been added before an issue moves to *in progress*, it is added while the issue is in this place. The team JAGUAR has an optional testing step before issues are *done*. Interestingly, about ten percent of the cases we observed involve issues moving back to *in test* from *done*, and the mean time until an issue is moved back is three days. It is unclear if this is an administrative mistake or whether these issues need rework.

Another interesting observation we derived from the mined process graphs is that, compared to bug-fixing processes, implementing a feature appears to include a deterministic path for deciding which issues may be *rejected*. For instance, in team BISON, issues that end up being *rejected* primarily originate from the middle *inbox* place, while other issues follow one of the four paths to completion (*done*). Similarly, in team JAGUAR, *rejected* issues are those primarily modified by architects (indicated by the red transition color involved with the topmost *to do* place) and those that originated from the topmost *inbox* place. Although we did not investigate the underlying reasons in depth, this observation suggests that there may be early predictors in an issue's lifecycle that determine whether a feature will be completed or not. Specifically, features with neither an assignee nor a fix version and features that appear immediately in *to do* instead of *inbox* are more likely to be *rejected* in both teams. In the future, TFS aims to explore such cases to understand the causes and ideally provide better automation to its developers.

V. DISCUSSION AND LEARNING

The technique we developed for mining processes from Jira issues worked as intended, providing insights into activities that are outside of typical version-control systems and helping TFS understand its development processes better. In Section IV, we exemplified insights we gained on the bug-fixing and feature-development processes of the 24 teams. Particularly, we were able to identify similarities and differences in terms of the issue states, general categories (e.g., missing testing), concrete workflows, and time it takes for issues to move on. As examples, we found that one team relies solely on customer-reported bugs while another team creates such reports themselves, or that simple bugs are resolved faster by some teams. For TFS, it has become clear that these differences have implications for organizing and improving the bug-fixing processes. Our results helped understand such deviations and motivated the need for further analyses. For instance, a missing initial state may simply be a consequence of the teams' respective components or because the developers of that team do not create issues for their own bug reports. Moreover, TFS can unify and thereby improve bug-fixing processes by analyzing why certain bugs are resolved much faster, for which our technique identified the starting points to look into.

Furthermore, we learned that in both bug-fixing processes we exemplified, the product owner plays a crucial role in determining when issues move to the next place. In contrast, within the feature-development processes, the product owner is mainly involved in backlog refinement when issues are still in places related to *inbox* or *to do*. Later in the feature-development process, engineers take over the administration. For all examples, we filtered out between 5 and 10 % of all Jira issues that did not conform to the general processes, indicating that teams generally adhere to a standard way of working. Still, they sometimes deviate from this way of working, and there are quite some differences between the teams. Here, our technique helps TFS understand who is involved in what steps, best practices, and rare corner cases (which we omit, but which can easily be added). Consequently, the graphs we create provide a valuable basis for aligning processes and identifying corner cases, which may require particular attention.

Regarding the time spent, in the bug-fixing processes, we found that issues spend most of their time in a place corresponding to *to investigate* or *to do*. The *to do* place is not an active one, but work is taking place as part of the investigation in the *to investigate* place. However, because there is no place "investigating" in the graphs (because it is not in the Jira issues), it is unclear how long an investigation takes place. Overall, *in progress* takes up only a small portion of time in the entire bug-fixing processes; most time is spent waiting for work to be done, as indicated by thicker arrows going into and out of *to do* places. The same applies to features where most time is spent on *to do* rather than *in progress*, where active work takes place. Interestingly, in terms of issue states, the feature-development processes seem simpler and more deterministic compared to the bug-fixing processes, despite the longer time spent on feature issues. This simplicity may be due to the fact that features typically have clear personnel assignments from the outside, whereas bugs often require investigation to determine the most suitable team member to work on them. Discussing these insights with TFS, we learned that our technique can help TFS understand potential bottlenecks, performance improvements, and idle times in development processes, which can then be further analyzed and potentially resolved.

Lastly, we compared the output of our technique to the output produced by two established process-mining tools, Disco [10] and ProM [36]. Disco uses a fuzzy miner, which creates a single place per action, resulting in numerous places connected to many other places because every action can happen while a Jira issue is in any place. For instance, when an assignee is added while the issue is in the status *to do* or *in test*, the action leads to an equivalent place in the graph. ProM uses an inductive miner, which contains a single starting and ending place. This algorithm can display only one process at a time and does not show self-transition loops, leading to many optional actions in a chain in the resulting Petri Net. Compared to both techniques, we followed a different idea and mine process graphs with more details that yield different, novel insights.

Learning Summary

We learned that our technique works as intended; it yields novel insights compared to the related work and is perceived as highly valuable by TFS. This underpins the value of mining processes from Jira issues for developers and organizations to understand and improve their development as well as maintenance processes.

VI. Threats to Validity

Internal Validity. The reliability of our technique depends on how reliably Jira issues are used within an organization or developer community. For instance, the measured time may vary if Jira issues are not updated promptly, the processes may be incorrect if the labels are incorrect, and anything that is not documented in Jira cannot be mined. Concretely, this impacts our evaluation at TFS since some developer teams we investigated updated issues only during their daily stand-up meetings. However, this means that all times for those teams are equally expanded, and we considered such properties when analyzing the mined process graphs to mitigate the consequent threat to the internal validity.

External Validity. We designed our technique for Jira issues specifically and evaluated it in one company only. Consequently, our findings may not be generalizable to developers in other organizations or in open-source communities. To improve the external validity and mitigate this threat, we conducted a multi-case study. While all teams are part of the same company, they work independently and, as we have shown, they employ different processes. For this reason, we argue that this threat is mitigated by conducting our evaluation with multiple independent teams and developers.

VII. Related Work

Traditionally, process discovery focuses on discovering relationships between activities from sequences of activity executions, where Petri Net places encode control-flow logic [1]. Discovering object-centric Petri Nets [35] allows distinguishing different data objects' sequences of activity executions explicitly, while places only model control-flow logic. Conversely, discovering a set of inter-linked state-based models (transition systems) over multiple objects from state-based event logs [37] allows discovering places directly, but actions are discovered implicitly and cannot be visually annotated. Furthermore, such a model is difficult to understand and explore. In contrast, we explicitly mine state and action information to construct a process graph covering both aspects directly.

Various researchers have applied process mining to software repositories [3], [7], [13], [19], [26], [29]. Process mining involves retrieving business processes from event logs produced by information systems. For instance, Poncin et al. [36] proposed a framework for analyzing software repositories using ProM, a tool commonly used in process mining. This tool allows mining processes from multiple repositories and combining events occurring in different places to create a single event log. Marques et al. [18] used process mining on Jira history logs with ProM and found that the α-, heuristic-, and inductive-miners produced a spaghetti model or an incorrect model. The authors settled on using a fuzzy miner, which produced a simpler model that only included state transitions. Most closely related to our study are the works by Gupta and Sureka [12] as well as Juneja et al. [14], who consider issue-trackers as an information source for their mining techniques. There are some overlaps between these two works and ours that improve our confidence in our technique and the validity of our case study, for instance, regarding the identification of process deviations in real-world software development. In contrast to existing techniques, we build on a novel strategy for analyzing and interpreting issues that yields more in-depth insights, particularly with respect to the involved developer roles—which have not been integrated into previous works. So, our technique complements existing research on process mining. z

VIII. Conclusion

In this paper, we have proposed and evaluated a technique for mining software development and maintenance processes from Jira issues. Our technique iterates through all provided issues and recovers processes based on the available states and version histories. If such fields are used and maintained, our technique can help understand the processes developers employ to potentially improve them. We conducted a multi-case study in which we used our technique to mine processes for 24 developer teams at TFS, a large international company. The results show several interesting differences in the processes employed and highlight what activities the company may want to investigate in the future. Upon inspecting and discussing these findings, we received very positive feedback from TFS regarding the usefulness of our technique. Consequently, we extend the state-of-the-art with a complementary process-mining technique for Jira issues that can deliver novel insights and advances upon the related work.

In the future, we aim to propose techniques for automatically analyzing and comparing the process graphs we mined. For instance, it can be helpful for developers to have highlights for major differences, bottlenecks, or rare deviations. Additionally, we plan to extend our technique by considering more Jira fields to add further information to the graphs and by developing interactive visualizations. This aligns with the need to quality-check the Jira issues used by our technique, for instance, to deal with missing values or to ensure that the provided data is correct.

References

[1] A. Augusto, R. Conforti, M. Dumas, M. La Rosa, F. M. Maggi, A. Marrella, M. Mecella, and A. Soo, "Automated Discovery of Process Models from Event Logs: Review and Benchmark," *IEEE Transactions on Knowledge and Data Engineering*, vol. 31, no. 4, pp. 686–705, 2019.

[2] H. Balci and U. Dogrusoz, "fCoSE: A Fast Compound Graph Layout Algorithm with Constraint Support," *IEEE Transactions on Visualization and Computer Graphics*, vol. 28, no. 12, pp. 4582–4593, 2021.

[3] J. Caldeira and F. Brito e Abreu, "Software Development Process Mining: Discovery, Conformance Checking and Enhancement," in *International Conference on the Quality of Information and Communications Technology (QUATIC)*. IEEE, 2016, pp. 254–259.

[4] S. Cha, R. N. Taylor, and K. Kang, Eds., *Handbook of Software Engineering*. Springer, 2019.

[5] B. Coremans, A. L. Klomp, S. A. Rukmono, J. Krüger, D. Fahland, and M. R. V. Chaudron, "Process Mined from Jira Issues at a Large Company," 2023. [Online]. Available: https://doi.org/10.5281/zenodo.8275679

[6] U. Dogrusoz, E. Giral, A. Cetintas, A. Civril, and E. Demir, "A Layout Algorithm for Undirected Compound Graphs," *Information Sciences*, vol. 179, no. 7, pp. 980–994, 2009.

[7] C. dos Santos Garcia, A. Meincheim, E. R. F. Junior, M. R. Dallagassa, D. M. V. Sato, D. R. Carvalho, E. A. P. Santos, and E. E. Scalabrin, "Process Mining Techniques and Applications – A Systematic Mapping Study," *Expert Systems with Applications*, vol. 133, pp. 260–295, 2019.

[8] A. Dresch, D. P. Lacerda, and J. A. V. Antunes Jr., *Design Science Research*. Springer, 2015.

[9] A. Fuggetta and E. Di Nitto, "Software Process," in *Conference on the Future of Software Engineering (FOSE)*. ACM, 2014, pp. 1–12.

[10] C. W. Günther and A. Rozinat, "Disco: Discover Your Processes," in *International Conference on Business Process Management (BPM)*. CEUR-WS.org, 2012, pp. 40–44.

[11] C. W. Günther and W. M. P. van der Aalst, "Fuzzy Mining – Adaptive Process Simplification Based on Multi-Perspective Metrics," in *International Conference on Business Process Management (BPM)*. Springer, 2007, pp. 328–343.

[12] M. Gupta and A. Sureka, "Process Cube for Software Defect Resolution," in *Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2014, pp. 239–246.

[13] S. Jaqueline Urrea-Contreras, B. L. Flores-Rios, M. Angélica Astorga-Vargas, and J. E. Ibarra-Esquer, "Process Mining Perspectives in Software Engineering: A Systematic Literature Review," in *Mexican International Conference on Computer Science (ENC)*, 2021, pp. 1–8.

[14] P. Juneja, D. Kundra, and A. Sureka, "Anvaya: An Algorithm and Case-Study on Improving the Goodness of Software Process Models Generated by Mining Event-Log Data in Issue Tracking Systems," in *Annual Computer Software and Applications Conference (COMPSAC)*. IEEE, 2016, pp. 53–62.

[15] J. Klünder, R. Hebig, P. Tell, M. Kuhrmann, J. Nakatumba-Nabende, R. Heldal, S. Krusche, M. Fazal-Baqaie, M. Felderer, M. F. Genero Bocco, S. Küpper, S. A. Licorish, G. López, F. McCaffery, Ö. Özcan Top, C. R. Prause, R. Prikladnicki, E. Tüzün, D. Pfahl, K. Schneider, and S. G. MacDonell, "Catching up with Method and Process Practice: An Industry-Informed Baseline for Researchers," in *International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 2019, pp. 255–264.

[16] J. Krüger, W. Mahmood, and T. Berger, "Promote-pl: A Round-Trip Engineering Process Model for Adopting and Evolving Product Lines," in *International Systems and Software Product Line Conference (SPLC)*. ACM, 2020, pp. 2:1–12.

[17] J. Krüger, M. Mukelabai, W. Gu, H. Shen, R. Hebig, and T. Berger, "Where is My Feature and What is it About? A Case Study on Recovering Feature Facets," *Journal of Systems and Software*, vol. 152, pp. 239–253, 2019.

[18] R. Marques, M. Mira da Silva, and D. R. Ferreira, "Assessing Agile Software Development Processes with Process Mining: A Case Study," in *Conference on Business Informatics (CBI)*, 2018, pp. 109–118.

[19] C. Mayr-Dorn, J. Tuder, and A. Egyed, "Process Inspection Support: An Industrial Case Study," in *International Conference on Software and Systems Process (ICSSP)*. ACM, 2020, pp. 81–90.

[20] B. Meyer, *Agile!* Springer, 2014.

[21] L. Montgomery, C. Lüders, and W. Maalej, "An Alternative Issue Tracking Dataset of Public Jira Repositories," in *International Coference on Mining Software Repositories (MSR)*. ACM, 2022, pp. 73–77.

[22] M. Mortada, H. M. Ayas, and R. Hebig, "Why do Software Teams Deviate from Scrum? Reasons and Implications," in *International Conference on Software and Systems Process (ICSSP)*. ACM, 2020, pp. 71–80.

[23] M. Ortu, G. Destefanis, B. Adams, A. Murgia, M. Marchesi, and R. Tonelli, "The JIRA Repository Dataset: Understanding Social Aspects of Software Development," in *International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)*. ACM, 2015, pp. 1:1–4.

[24] M. Ortu, G. Destefanis, M. Kassab, and M. Marchesi, "Measuring and Understanding the Effectiveness of JIRA Developers Communities," in *International Workshop on Emerging Trends in Software Metrics (WETSoM)*. IEEE, 2015, pp. 3–10.

[25] K. Pohl, G. Böckle, and F. J. van der Linden, *Software Product Line Engineering*. Springer, 2005.

[26] W. Poncin, A. Serebrenik, and M. van den Brand, "Process Mining Software Repositories," in *European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2011, pp. 5–13.

[27] N. Prenner, C. Unger-Windeler, and K. Schneider, "How are Hybrid Development Approaches Organized? - A Systematic Literature Review," in *International Conference on Software and Systems Process (ICSSP)*. ACM, 2020, pp. 145–154.

[28] N. Prenner, C. Unger-Windeler, and K. Schneider, "Goals and Challenges in Hybrid Software Development Approaches," *Journal of Software: Evolution and Process*, vol. 33, no. 11, pp. e2382:1–25, 2021.

[29] V. Rubin, C. W. Günther, W. M. P. van der Aalst, E. Kindler, B. F. van Dongen, and W. Schäfer, "Process Mining Framework for Software Processes," in *International Conference on the Software Process (ICSP)*. Springer, 2007, pp. 169–181.

[30] M.-A. Storey, N. A. Ernst, C. Williams, and E. Kalliamvakou, "The Who, What, How of Software Engineering Research: A Socio-Technical Framework," *Empirical Software Engineering*, vol. 25, no. 5, pp. 4097–4129, 2020.

[31] C. Sürücü, B. Song, J. Krüger, G. Saake, and T. Leich, "Establishing Key Performance Indicators for Measuring Software-Development Processes at a Large Organization," in *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2020, pp. 1331–1341.

[32] ——, "Using Key Performance Indicators to Compare Software-Development Processes," in *Software Engineering (SE)*. GI, 2021, pp. 105–106.

[33] W. M. P. van der Aalst and C. Stahl, "Modeling Business Processes – A Petri Net-Oriented Approach," Master's thesis, Eindhoven University of Technology, 2011.

[34] W. M. P. van der Aalst, *Process Mining: Data Science in Action*. Springer, 2016.

[35] W. M. P. van der Aalst and A. Berti, "Discovering Object-Centric Petri Nets," *Fundamenta Informaticae*, vol. 175, no. 1-4, pp. 1–40, 2020.

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

[37] M. L. van Eck, N. Sidorova, and W. M. P. van der Aalst, "Guided Interaction Exploration and Performance Analysis in Artifact-Centric Process Models," *Business & Information Systems Engineering*, vol. 61, no. 6, pp. 649–663, 2019.

[38] R. K. Yin, *Case Study Research and Applications: Design and Methods*. Sage, 2018.