Nirikshan: Mining Bug Report History for Discovering Process Maps, Inefficiencies and Inconsistencies

Monika Gupta  Ashish Sureka
Indraprastha Institute of Information Technology, Delhi (IIITD)
New Delhi, India
{monikag, ashish}@iiitd.ac.in

ABSTRACT
Issue tracking system such as Bugzilla, Mantis and JIRA are Process Aware Information Systems to support business process of issue (defect and feature enhancement) reporting and resolution. The process of issue reporting to resolution consists of several steps or activities performed by various roles (bug reporter, bug triager, bug fixer, developers, and quality assurance manager) within the software maintenance team. Project teams define a workflow or a business process (design time process model and guidelines) to streamline and structure the issue management activities. However, the runtime process (reality) may not conform to the design time model and can have imperfections or inefficiencies. We apply business process mining tools and techniques to analyze the event log data (bug report history) generated by an issue tracking system with the objective of discovering runtime process maps, inefficiencies and inconsistencies. We conduct a case-study on data extracted from Bugzilla issue tracking system of the popular open-source Firefox browser project. We present and implement a process mining framework, Nirikshan, consisting of various steps: data extraction, data transformation, process discovery, performance analysis and conformance checking. We conduct a series of process mining experiments to study self-loops, back-and-forth, issue reopen, unique traces, event frequency, activity frequency, bottlenecks and present an algorithm and metrics to compute the degree of conformance between the design time and the runtime process.

Categories and Subject Descriptors
H.4 [Information Systems Applications]: Miscellaneous; D.2.8 [Software Engineering]: Metrics—complexity measures, performance measures

General Terms
Algorithms, Experimentation, Measurement

Keywords

1. RESEARCH MOTIVATION AND AIM
Issue Tracking Systems (ITS) such as Bugzilla\(^1\), Mantis\(^2\) and JIRA\(^3\) are Process Aware Information Systems (PAIS) used during software development and maintenance for issue (feature enhancement requests and defects) tracking and management. Large and complex software projects, both open-source and commercial, define a bug lifecycle with different states and transitions for a bug report and a process model (activities, events, actors, transitions and workflow) serving as guidelines to the project team. ITS such as Bugzilla logs events generated during a bug lifecycle (such as who reported the bug and when, change of resolution and status field of a bug, developer and component assignment or reassignment) and refers to the event logs as Bug History. Process Mining (intersection of Business Process Management and Data Mining) consists of mining event logs generated from business process execution supported by a PAIS \(^{[10]}\). The focus of the research presented in this paper is on the application of process mining techniques to mine bug history data and event logs generated from an ITS (software repository). Issue tracking and management is an activity which is integral to software development and maintenance and is fraught with several challenges. The work presented in this paper is motivated by the need to mine process data generated by ITS to extract insights particularly inefficiencies and inconsistencies. This can be useful to practitioners for solving problems resulting in productivity and efficiency improvements in software maintenance. The specific research aims of the work presented in this paper are the following:

1. To develop a generic algorithm for quantitatively measuring the compliance (conformance checking and verification) between the design time and the runtime (reality) process model using metrics in the context of issue tracking and management.

2. To investigate the application of process mining platforms such as Disco\(^4\) and state-of-the-art algorithms for discovering process maps from event logs (in our case, bug re-

---

1. https://bugzilla.mozilla.org/
port history data) and then analyzing the discovered process map for understanding the actual runtime process.

3. To define a process mining framework and conduct a case-study on popular open-source projects like Firefox and Core from Mozilla foundation for analyzing the performance of various process-related activities and identifying inefficiencies (statistics on events, activities and transitions) and imperfections (opening of bugs, bottlenecks, self-loops, back-and-forth between activities).

2. RELATED WORK AND RESEARCH CONTRIBUTIONS

In this section we discuss closely related work (to the research presented in this paper) and present the novel research contribution of this paper in context to the existing work. We conduct a literature review in the area of process mining software repositories. Table 1 presents a list of 7 closely related works arranged in a chronological order (year 2006 to 2012). We synthesize existing work and classify the papers in terms of attributes such as software repository (Version Control Systems, Defect Tracking Systems, Mail Archives), project (ArgoUML, Linux, Proprietary), OSS or CSS and research objective. The most closely related research to the work presented in this paper is by Sunindoyo et. al. [9] in which they proposed a framework for effective engineering process observation in OSS projects. They used hypothesis testing approach to verify design model with runtime event log from bug history data. They obtained process map for RHET bug history data using heuristic mining algorithm of process mining tool ProM. The conformance verification is not in-depth and done for few dimensions by proving/disproving the hypothesis. Poncin et. al. [5] combined the information from different repositories by matching related events. This is achieved by a prototype FRASR (FRamework for Analyzing Software Repositories). The resulting log is analyzed using ProM dotted chart visualization and fuzzy miner plugin for role classification and fuzzy graph generation (actual view of bug lifecycle) respectively. However, major emphasis is on organizational perspective and not on analyzing runtime process.

From literature, we observe that major research focus is towards designing algorithm for process discovery, defining possible broad perspectives (such as process, cases and organizational) of event log analysis and its applicability. However, to the best of our knowledge the outcome of process perspective (discovered process map) for ITS event log (which is very important repository of a software project) is not analyzed from multiple dimensions. We hypothesize that there is a need to have an in-depth analysis to help process analyst in making informed decisions for the process improvement and effectively the overall stability of an organization.

In context to previous work, the study presented in this paper addresses the challenges by making following novel and unique research contributions:

1. To the best of our knowledge, the work presented in this paper is the first in-depth study on process mining event logs (bug life-cycle and history data) from issue tracking system (process aware information systems) for discovering runtime process maps (and statistics on activities, events, transitions, unique traces), inefficiencies (issue reopens, bottlenecks, self-loops, back-and-forth between activities) and inconsistencies (an algorithm to compute degree of conformance between design time and runtime process models).

2. A case-study and empirical analysis (implementing the process mining framework and conformance checking algorithm) on issue tracking system data of Firefox and Core products (large, complex, long-lived and popular open-source projects) from Mozilla Foundation. We present new results and fresh perspectives (by mining large real-world data) motivating the need and demonstrating the usefulness of process mining event log data or traces generated from issue tracking systems.

3. RESEARCH METHODOLOGY AND FRAMEWORK

We conduct experiments on issue tracking system data from Firefox and Core open-source projects. Table 2 displays the experimental dataset details. We chose Firefox and Core for our analysis as these projects are large, complex and long-lived. Firefox is Mozilla's flagship software product which is the third most-used web browser for Windows, MAC and Linux. Core includes shared components used by Firefox and other Mozilla software. The bug report data for Firefox and Core projects are publicly available and hence our results can be replicated and used for comparison by other researchers. We conduct experiments on two projects (Firefox and Core) to remove one-project bias in our results and conclusions (increase the generalizability of our results). As shown in Table 2, the size of the experimental dataset is 12234 issue reports from Firefox and 24253 reports from Core (one year period: 1 January 2012 to 31 December 2012).

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>First issue creation date</td>
<td>1 Jan 2012</td>
</tr>
<tr>
<td>Last issue creation date</td>
<td>31 Dec 2012</td>
</tr>
<tr>
<td>Date of extraction</td>
<td>14 July 2013</td>
</tr>
<tr>
<td>Total issues in 2012</td>
<td>111234</td>
</tr>
<tr>
<td>Issues not authorized for access</td>
<td>15638</td>
</tr>
<tr>
<td>Issues without history</td>
<td>3149</td>
</tr>
<tr>
<td>Total issues extracted for Firefox</td>
<td>12234</td>
</tr>
<tr>
<td>Total issues extracted for Core</td>
<td>24253</td>
</tr>
<tr>
<td>Total activities for Firefox</td>
<td>40233</td>
</tr>
<tr>
<td>(including Reported)</td>
<td></td>
</tr>
<tr>
<td>Total activities for Core</td>
<td>88396</td>
</tr>
<tr>
<td>(including Reported)</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Experimental dataset details for Core and Firefox (open source Mozilla Project).

The complete framework (called as Nirikshan which is a Hindi word meaning inspection or review), as depicted in Figure 1, has three major stages: 1. data extraction, 2. data transformation, and 3. process mining.

1. Data extraction: We extract bug report history (refer to Figure 2 displaying the snapshot of a Bugzilla bug history page) using Bugzilla APIs (through XML-RPC or JSON-RPC interface). The bug report history serves as the process event log generated by the ITS. We extract Status, Resolution (only for closed), Assignee, QA Contact and Component from bug history to derive the process map. In our experiments, we record creation timestamp for the activity Reported.
Table 1: Literature survey of papers at the intersection of mining software repositories and process mining arranged in chronological order.

<table>
<thead>
<tr>
<th>Author(s)</th>
<th>Year</th>
<th>Software Repository</th>
<th>Project</th>
<th>OSS/CSS</th>
<th>Objective</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kindler et. al.</td>
<td>[3]</td>
<td>SCM</td>
<td>Dummy versioning log to make it independent of SCM system</td>
<td>OSS</td>
<td>Proposed incremental workflow mining algorithm to semi-automatically derive process model from versioning information.</td>
</tr>
<tr>
<td>Wijters et. al.</td>
<td>[10]</td>
<td>2006</td>
<td>Dummy event log with random traces</td>
<td>NA</td>
<td>Distinguished three mining perspectives and focussed on the process perspective. Described and analysed HeuristicsMiner algorithm</td>
</tr>
<tr>
<td>Rubin et. al.</td>
<td>[7]</td>
<td>2007</td>
<td>SCM, VCS</td>
<td>AgroUML OSS</td>
<td>Linear Temporal Logic (LTL) Checking (Conformance Analysis), Social Network Discovery, Performance Analysis, Petri Net Discovery</td>
</tr>
<tr>
<td>Akman et. al.</td>
<td>[1]</td>
<td>2009</td>
<td>SCM</td>
<td>Industry project (for Turkish Armed Forces)</td>
<td>Analyzed and compared the effectiveness of four process discovery algorithms on software process. Analyzed discrepancies between real time and design time process.</td>
</tr>
<tr>
<td>Knab et. al.</td>
<td>[4]</td>
<td>2010</td>
<td>Issue Tracking System</td>
<td>Industrial Project (EUREKA/ITEA project SERIOUS)</td>
<td>Interactive approach to visualise effort estimation and process lifecycle patterns in ITS to detect outliers, flaws and interesting properties.</td>
</tr>
<tr>
<td>Puncin et. al.</td>
<td>[5]</td>
<td>2011</td>
<td>Bug repositories, mail archives, SVM</td>
<td>aMSN, GCC OSS</td>
<td>Combined different repositories for analysis using a prototype, FRASR. Role classification and Bug life cycle construction using ProM.</td>
</tr>
</tbody>
</table>

For open bugs, status can be New, Unconfirmed, Assigned and Reopened which is captured as activity. For closed bugs, Verified status is captured as it is while for the status Resolved, the resolution is captured as activity. The resolution can be Fixed, Invalid, Wont-fix, Duplicate, Worksforme and Incomplete. We consider every closed resolution as a unique event for detailed analysis as it will also capture the transitions between different resolutions. Assignee, QA Contact and Component are recorded as Dev-reassign, QA-reassign and Comp-reassign respectively as they are important events in a bug lifecycle and can lead to change in state like a bug should be marked New after developer reassignment. Based on the mentioned significance of captured activities, we hypothesize that they are important for good characterization of bug’s control-flow (the tasks performed during the progression of a bug). They cover the tasks performed starting from inception to closing of bug. We obtain timestamp corresponding to an activity from the when field of bug history as can be seen in Figure 2 to order the activities in the sequence of their actual execution (while generating the process map via Disco). The performer of the activity is treated as resource and is captured from the who field of bug history.

2. Data transformation: One of the major challenges in software process mining is to produce a log conforming to the input format of process mining tool [5]. Therefore, data preprocessing is very crucial before analysis in a process mining tool. Following is the processing performed to address the challenges:

- Activities with timestamp, bug ID and associated resource make the event log for process map generation.
- Reported event is not stored in bug history so it is added to the event log with creation timestamp.
- Initial state is not captured in history if no activity leading to change in status and resolution is performed. For such cases only Reported (with creation timestamp) is stored in event log and not the initial state. Otherwise, initial state is also stored with the same timestamp as Reported. For instance, the process map from the history of a bug is shown in Figure 2.
- There are cases when Comp-reassign, Dev-reassign and QA-reassign have same the timestamp with other activity. This “same timestamp conflict” is resolved by subtracting delta from the timestamp of these activities such that the final ordering in all conflicts (except Reopen) should be: Comp-reassign → QA-reassign → Dev-reassign → Other activity

This is because component assignment triggers

\[\text{http://www.bugzilla.org/docs/2.18/html/lifecycle.html} \]

\[\text{https://bugzilla.mozilla.org/page.cgi?id=fields.html#status} \]
Figure 1: **Nirikshan**: Proposed research framework with three major phases as 1. data extraction, 2. data transformation, and 3. process mining.

Figure 2: A snapshot of Mozilla bug report history (event log) and corresponding process map.

devender and QA assignment\(^7\) followed by activities for bug progression. However, *Reopen* is the reactivation of a bug rather than filing a new bug so it is resolved as: *Reopen* $\rightarrow$ *Comp-reassign* $\rightarrow$ *QA-reassign* $\rightarrow$ *Dev-reassign*.

3. **Process mining**: The event log can be mined from various perspectives\(^10\). We perform process mining to discover runtime process model (process perspective), analyze performance and verify conformance of as-is model with the design time process model. We analyze statistics for activity, transitions, events and unique traces from discovered process map. The performance analysis is done for reopening of bugs, presence of loops, back-forth transitions and identifying bottlenecks.

4. **PROCESS DISCOVERY**

There are many process mining tools such as ProM (open source) and Disco\(^8\) (commercial) used to obtain process model. We import preprocessed data into Disco to obtain process map and other statistical information for Core and Firefox. Disco miner is based on the proven framework of the Fuzzy Miner with completely new set of process metrics and modeling strategies\(^9\). It is used to discover the runtime process from the actual event log generated during the progression of a bug. Bug ID is selected as case (for process map generation in Disco) to associate all activities pertaining to the same bug ID and hence, we can visualize the lifecycle of a bug. We have 15 nodes each corresponding to an activity in the process map for both the projects and the directed edge represents a transition between two activities (node). With 15 nodes, we can have a maximum of 210 unique transitions including self-loop given *Reported* as a source and hence no inward arrow possible. The process map obtained is fairly complex at the highest resolution (including infrequent transitions) with 160 and 156 unique transitions for Core and Firefox respectively. For clarity, the process map at 20% resolution (core transitions) for Firefox is shown in Figure 3. Label of each edge indicates the absolute frequency of transition. The shade and thickness corresponds to the frequency with more frequent being dark and less frequent as light.

4.1 **Activity Frequency Analysis**

The relative occurrence of all activities with absolute frequency is shown in Table 3. After *Reported* (exists for all the cases), the most frequent event is *Unconfirmed* for Firefox and *New* for Core. This reduces the confirmation efforts for reported bugs in Core. Figure 4 shows the distribution of activities over cases, that is, how much percentage of cases have the given activity in their event log.

As evident from Figure 4, a good percentage of cases have *Dev-reassign* and *Comp-reassign* event in lifecycle, also their relative frequency is high which is undesired. We see that QA reassignment is happening quite often. Developer reassignment is much higher for Core than Firefox. It leads to the wastage of developer’s time and efforts and delay bug fixing. Hence the efforts should be made to minimize reassignments. A bug once resolved should be verified. However, only around 4% of bugs are verified by the QA manager for both the projects which is significantly less than the resolved

---

\(^7\)http://www.bugzilla.org/docs/2.20/html/components.html  
\(^8\)Disco is a proprietary tool for which we availed academic license and used for our analysis.  
bugs signalling inefficiency. It highlights the need to uncover reasons for infrequent verification and address them.

Further, we can clearly observe from Figure 4 that Core has almost double chances of getting a bug Fixed which is the recommended situation. While for Firefox, comparatively a large number of bugs are marked Duplicate, Invalid and Worksforme which means a lot of practitioner’s time is going in addressing issues less useful for overall product quality improvement. To avoid duplicates, an efficient search mechanism can be used before reporting a bug or an extension to give “Duplicate bug” warning at the time of bug submission. Similarly, clarity on what should be reported as bug to reduce invalid bugs and emphasize need for clear description of bugs for better understanding.

4.2 Transition Frequency Analysis

Intuitively, frequent transitions should be the part of design process. We find that frequent transitions are usually part of lifecycle defined for Bugzilla. However, there are some which are infrequent even though allowed in design time process namely, Verified → Reopen with only 14 and 4 cases, Reopen → Assigned with 20 and 8 cases for Core and Firefox respectively. Sometimes this may be because of the fact that source or destination activity is less frequent. Other reason can be that majority of the cases have transition from a given state to a specific state, so the frequency for other possible transitions become low and hence gives an impression that it is not part of design process. The infrequent occurrence of Assigned → New transition implies that there is a non-adherence to defined process, that is, a bug should be marked New after Dev-reassign but it is not practiced always as the frequency of developer reassignment for both the projects. Also, Verified
1. Reopen Analysis

Reopened bugs\(^{10}\) increase the maintenance costs, degrade overall user-perceived quality of the software and lead to unnecessary rework by busy practitioners [8]. If fair number of fixed bugs are reopened, it could indicate instability in the software system [11]. Bug reopening has a significant importance in both open source and commercial software systems. Understanding bug reopening is of significant interest to the practitioner’s community in order to characterize actual quality of the bug fixing process [11]. Therefore, the analysis of reopened bugs will help process owner to take preventive actions to minimize the reopening of bugs.

For both the products, of all the labels, Wontfix is more likely to be reopened in comparison with others as shown in Figure 6a. Also there are very few cases reopened after QA-reassignment, component reassignment and developer reassignment. Shihab et. al. [8] showed that the last status of the closed bug is an important influential factor for reopening. The reasons [11] [8] for reopening of the closed bug with given label could be:

- Wontfix/Invalid/Incomplete/Worksforme: Clear steps to reproduce and additional information becomes avail-

\(^{10}\)According to Bugzilla@Mozilla, the bug was once resolved, but the resolution was deemed incorrect.
able to better understand reported bug and determine root cause. Also, if the priority of bug is underestimated and hence closed.

- **Duplicate**: A bug accidentally marked duplicate due to similar symptoms or title matching with the existing bug without proper understanding of the root cause.

- **Fixed**: Incompletely or incorrectly fixed bugs due to poor root cause understanding, regression bug (the bug reappears in the new version) or the boundary cases missed by developer during testing.

- **Verified**: Incorrectness realized later or extra information becomes available which triggers reopening of bug.

Figure 6a shows percentage of bugs getting reopened for all the possible closed states. The interpretations from the figure are as follows:

- Comparatively high percentage of bugs closed with label *Wontfix, Invalid, Incomplete* and *Workforme* are getting reopened for Core. To improve the quality and understanding, more efforts should be made to ensure that sufficient information is retrieved from the reporter before closing. This can be done in two ways: 1. being interactive and clarifying things with the reporter after a bug has been reported, and 2. improve initial template to capture sufficient details beforehand, to reduce the delay. Disagreement in the priority of bugs should be minimized by defining clear guidelines to decide priority.

- We see that the chances of getting a *Duplicate* bug reopened in Core (around 5%) are more than double of Firefox (around 2%). Thorough understanding of bug should be done before marking a bug *Duplicate*. The decision should not be based on superficial attributes.

- After *Wontfix*, bugs resolved as *Fixed* are prone to reopening in Firefox (6%) unlike Core (with Workforme). A proper understanding of root cause is necessary for fixing a bug. Assumptions related to reported bug should be avoided. If unclear, the owner should ask the reporter for clarification and necessary information. Minimal required testing of fixed bugs should be performed to avoid failures at later stages.

- There are very less chances of reopening for *Verified* bugs (1% or less). So, the QA manager should be encouraged to verify final resolution carefully.

Of all the reopened bugs, the major contribution is from *Fixed, Duplicate, Wontfix, Workforme* and *Invalid*. Rest all constitutes very less percentage as evident from Figure 6c for Firefox and Figure 6b for Core. Since the total number of *Fixed* bugs is high, even if less percentage of *Fixed* get reopened, it contributes more towards total reopened bugs. Hence the reasons for reopening of *Fixed* bugs should be dealt with high priority.

### 5.2 Self-loop and Back-forth Analysis

It is important to study recurrent loops as they are indicators of deeper problems. It is difficult to detect such ping-pong patterns where a bug is passed repeatedly without any progress [2]. Self-loop is an edge that starts and ends at the same node (activity). It shows consecutive repetition of the same activity. Presence of loops is undesired as it indicates imperfection. Diagonal elements of the confusion
matrix shown as Table 6 denotes absolute number of loops where first entry corresponds to Core and second to Firefox. The process map obtained for Core and Firefox have loop mainly for Comp-reassign, Dev-reassign and QA-reassign as observed in Table 6. For Verified, there are 9 and 3 cases for Core and Firefox respectively which is very less and can be ignored. Most of the cases have one loop, that is, the decision made in first attempt could be imperfect and needs correction so same activity repeated two times. Maximum loops are for developer reassignment (1296/222) and component reassignment (471/257). It is not easy to decide the component to which bug pertains and the person to whom the bug should be assigned. The number of loops are more than one in some cases which means a lot of time is going in making decision for right assignment and component identification. Loops cause the undesired delay which is as high as 50.25 days and 30.35 days for Firefox and Core respectively in case of developer reassignment as shown in Table 5.

<table>
<thead>
<tr>
<th>Activities</th>
<th>Core</th>
<th>Firefox</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dev</td>
<td>Mean: 50.2</td>
<td>Mean: 50.2</td>
</tr>
<tr>
<td></td>
<td>Median: 9.9</td>
<td>Median: 9.8</td>
</tr>
<tr>
<td></td>
<td>Min: 3.8</td>
<td>Min: 3.8</td>
</tr>
<tr>
<td></td>
<td>1Q: 13.0h</td>
<td>1Q: 9.7</td>
</tr>
<tr>
<td></td>
<td>3Q: 65.9</td>
<td>3Q: 2.6</td>
</tr>
<tr>
<td></td>
<td>Max: 437.8</td>
<td>Max: 19.4</td>
</tr>
<tr>
<td></td>
<td>SD: 84.8</td>
<td>SD: 41.5</td>
</tr>
</tbody>
</table>

Table 5: Duration (unit: days-default, hours (h), minutes (m), second (s)) for loops in given states where Min: Minimum, 1Q: 1st quartile, 3Q: 3rd quartile, Max: Maximum, SD: Standard Deviation.

Back-forth: A transition from a state A to another state B and then back to state A is defined as one back-forth loop. We observe back-forth phenomenon between pair of activities (also referred to as a ping-pong pattern in the literature) in the runtime process model. Table 6 shows results of our empirical analysis for identifying back-forth pattern. Each cell of the Table 6 matrix (except diagonal values) shows frequency of back-forth loop between activity pairs (14 × 14 matrix in which the row activity is state A and the column activity is state B of back-forth loop). We notice that Unconfirmed, Component-reassignment, Developer-reassignment, QA-reassignment and Reopen are activities which are frequently involved in back-forth pattern. Our experimental results show that a Fixed bug being reopened and then again resolved as Fixed is frequent (refer to Table 6: 466 times for Core and 114 times for Firefox) indicating regression bug or imperfection. A back-forth loop phenomenon for Invalid, Worksforme, Duplicate and Wontfix state with Reopened shows disagreement between developers. One explanation can be that the team members are not convinced with initial decision and reopened the bug resulting in loss of productivity and increasing the mean-time to repair. Back-forth between Unconfirmed and any resolution is permitted according to the pre-defined Bugzilla lifecycle and we also observe in as-is process.

5.3 Bottleneck Analysis

A bottleneck is defined as a specific part within the runtime process map which is relatively time consuming and reduces the overall performance of the end-to-end process. Bottleneck identification (most time consuming states, activities and transitions) from an as-is process model is an actionable information for the process owner (from the perspective of root-cause analysis and process improvement). We compute the mean-time for every transitions defined in the bug lifecycle and make the following observations:

1. Our analysis shows that for Firefox project, confirmation to New state takes 33.4 days (mean value). In contrast, it takes on an average 19.4 days for direct assignment (directly to Assigned state). Confirmation seems to be a bottleneck (and hence can be expedited as it is resulting in delays). We observe that for Core project, the mean time taken for confirmation to New is 9.3 days which is relatively much less than Firefox.

2. The time taken for the transition New→Assigned is 17.1 days and 17.4 days for Core and Firefox respectively. This means on an average, there is a delay of around 17 days in assignment of bug from New state.

3. An Assigned bug is labelled as New in-case of a developer reassignment event (restart). Experimental result shows that the time-elapsed between two states is 57.4 days for Firefox and 6.4 days for Core. This should be avoided by correct developer assignment in first attempt or quick reassignment should be done if wrongly assigned because one wrong assignment will delay resolution for Firefox by 57.4 days on an average.

4. Resolution of a bug as Duplicate from possible source state: Unconfirmed, Assigned (not for Core), Reopened and New is taking comparatively less time for both the projects indicating efficiency in duplicate bug report identification. For Firefox, it is effective also as the chances of reopening are fairly low (Figure 6a).

5. Experimental results show that identification of cases where resolution is Worksforme, Wontfix and Incomplete is more time consuming irrespective of source state from which resolution is made. We believe the actionable information is that more efficiency improvement is required in their identification. For instance, resolution from New to Worksforme and Wontfix is taking exceptionally long duration of 19.3 week and 31.1 week respectively for Firefox and 20.5 week and 81.7 days for Core. For Core, Assigned to Wontfix is taking even more, that is, 14.8 weeks.

6. For both the projects, resolving a bug as Fixed takes less time from Assigned as the developer has already taken possession and worked to fix it. When Fixed directly from Unconfirmed or New, it takes more time.

7. The chances of Verified bug getting reopened are very less and takes 8.3 and 18.9 days on an average to reopen verified bug for Firefox and Core respectively. However, for Firefox Wontfix, Fixed and Worksforme are more likely to be reopened and there is delay of 14.9, 13.9 and 45.6 days in reopening of bugs from respective states. For Core, there is comparatively more delay in reopening of bugs resolved as Worksforme, Wontfix and Incomplete (24.1, 18.3 and 38.1 days respectively) and have good chances of getting reopened. Therefore, the reasons for their reopening should be dealt with more priority.
Table 6: Loop and Back-forth confusion matrix where 1st entry corresponds to Core and 2nd to Firefox (blank if no occurrence).

6. CONFORMANCE TESTING

Conformance checking aims at detection of inconsistencies between design time process model and the model obtained for as-is process from runtime event log. We define metrics to measure fitness, that is, how well observed process complies with the control flow defined in the design time process model and the point of inconsistency. We believe that it can be useful for identification and removal of obsolete parts which demands extra maintenance efforts [6].

We propose an algorithm 1 to evaluate fitness metric. We find number of cases with valid traces (only defined transitions) and its ratio with total cases to measure the extent of fitness. Event log and adjacency matrix, \( A \) (with row as source state and column as destination state, that is, \( 15 \times 15 \) in our case) has 1 in the cell if transition is permitted otherwise 0, are given as input. We obtain the event trace (array with states from event log in sequential order of their occurrence) for each case ID as shown in step 4 of algorithm 1. For optimization, we identify unique traces and count frequency of each. It is useful as most of the cases have same trace (refer Table 4) and we need not validate each case individually. Each unique trace is verified with adjacency matrix \( A \) for conformance. If it has all permitted transitions then the valid bit \( V \), is assigned value 1 else 0. If the evaluated value of fitness metric (refer to line 21 of algorithm 1) is less than 1 then there is deviation from defined model. To detect the cause of inconsistency, \( \text{inconsistentDetector()} \) (algorithm 2) is invoked which takes event log and adjacency matrix, \( A \) as input and returns inconsistency metrics. We create footprint matrix (refer step 10 of algorithm 2) of runtime event log and compare with design model (adjacency matrix) to identify inconsistent transitions (non-zero elements in \( ITF \)). All states (15 for our case) are stored in an array, \( \text{state} \). The frequency of transition between each pair of states is counted from event log and stored in Transition Frequency, \( TF \) matrix. Total inconsistent transitions are evaluated by adding the elements of \( ITF \) (in line 14 of algorithm 2). We evaluate the frequency (maximum element of \( ITF \)) and most frequent inconsistent transition in step 15 and 16 of algorithm 2.

To perform experiments, we consider only closed cases till last resolution state. Since we consider more activities than defined in bugzilla lifecycle, we define a design model (adjacency matrix) with extra states based on practitioner’s inputs and our understanding. The value of fitness metric, \( FM \) using algorithm 1 is 0.86 and 0.91 for Core and Firefox respectively. It clearly shows that Firefox has high conformance as compared to Core. However, both the projects have good compliance with defined design model. 818 traces out of 1164 total unique traces are valid for Core. Similarly, 403 traces out of 622 are valid for Firefox which implies that invalid traces usually have low frequency. As the \( FM < 1 \) for both the projects, we analyze cause for non-conformance, that is, detection of undefined transitions happening in runtime process. The proposed algorithm 2 is invoked resulting in 2412 and 739 total inconsistent transitions for Core and Firefox respectively. Surprisingly, the most frequent inconsistent transition is \( \text{Reported} \rightarrow \text{Assigned} \) for both the projects with frequency of 1643 and 305 respectively.

7. CONCLUSIONS

The runtime process map (reality or actual process model) derived from process mining event logs of 12234 Firefox and 24253 Core issue reports, reveal state transitions and event traces which are common and also the traces which are inconsistent with the design time process model. Empirical analysis reveals the distribution of 15 different activities and we infer that a statistically significant percentage of bug reports undergo component, developer and QA manager reassignment. Data analysis shows certain infrequent transitions (such as \( \text{Verified} \rightarrow \text{Reopen} \) and \( \text{Assigned} \rightarrow \text{New} \)) which can be used as guide to correct issues with the runtime process. We infer that majority of the bug lifecycle has 3 − 4 events while several bug reports have more than 7 events in lifecycle. The number of unique traces from start to finish is large (1164 and 622 variants for Core and Firefox respectively) and the top 2 unique trace constitutes more than 25% and 33% of the Firefox and Core bug reports respectively. Empirical analysis reveals the extent of transition from various states (such as \( \text{Worksforme}, \text{Wontfix}, \text{Verified}, \text{Invalid}, \text{Incomplete}, \text{Fixed} \) and \( \text{Duplicate} \)) to \( \text{Reopened} \) and shows that several \( \text{wontfix} \) and \( \text{worksforme} \) bug reports are getting reopened. We observe self-loop and back-and-forth transitions (undesirable and inefficient) and notice a back-forth pattern for the states: \( \text{Unconfirmed}, \text{Comp-reassign}, \text{Dev-reassign}, \text{QA-reassign} \) and \( \text{Reopened} \). We identify bottlenecks to help process owner for improvement. We propose a conformance...
Algorithm 1 evaluateFitnessMetric()

Require: Event log, Adjacency matrix A
Ensure: Fitness measure
1: while not at the end of Event log do
2: ∀ entries with Case ID i
3: if \( t_{SC} > t_{SM} > t_A > t_{Reported} \) then
4: Trace, \( T_i = \{Reported,A,B,C\} \);
5: end if
6: end while
7: Count frequency of each unique trace \( UT; \) as \( F_i \);
8: while \( \exists UT \) do
9: \( m := \) size of trace;
10: \( p := UT[1]; q := UT[2]; \) valid bit for \( UT; \) \( V_i = 1 \);
11: while \( p < m \) do
12: if \( A[p][q] == 1 \) then
13: \( p++; \)
14: \( q++; \)
15: else
16: \( V_i = 0; \)
17: break;
18: end if
19: end while
20: end while
21: Calculate Fitness metric:
\[
FM = \frac{\sum_{i=1}^{N}(F_i + V_i)}{\sum_{i=1}^{N}(F_i)}
\]
where \( N = \)Number of unique traces.
22: if \( FM < 1 \) then
23: inconsistentDetector(Event log, Adjacency matrix A);
24: else
25: No inconsistency;
26: end if

Algorithm 2 inconsistentDetector()

Require: Event log, Adjacency matrix A
Ensure: Inconsistent transition metrics
1: Array of states, \( state = \{state_1, state_2, ..., state_f\} \);
2: for \( i = state[1] : state[f] \) do
4: count = 0;
5: while not end of Event log do
6: if transition, \( i \rightarrow j \) then
7: \( \text{count}++; \)
8: end if
9: end while
10: Transition Frequency, \( TF[i,j] = \text{count}; \)
11: end for
12: end for
13: Inconsistent Transition Frequency matrix, ITF:
\[
= (TF - TF \circ A)
\]
14: Total Inconsistent Transition:
\[
= \sum (TF - TF \circ A)
\]
15: Highest frequency of inconsistent transition:
\[
= \text{max}(TF - TF \circ A)
\]
16: Most frequent inconsistent transition:
\[
= \text{argmax}(TF - TF \circ A)
\]

checking algorithm (between the design time and the runtime model) and the value of fitness metric discovered is 0.86 and 0.91 for Core and Firefox respectively.

8. ACKNOWLEDGEMENT

The work presented in this paper is supported by Prime Minister’s Fellowship awarded to the first author. The author would like to acknowledge SERB, CII and Infosys Limited for their support. The authors would like to thank Dr. Srinivas Padmanabhamuni for his valuable inputs.

9. REFERENCES